HTTP/2 в действии

HTTP (Hypertext Transfer Protocol – протокол передачи гипертекста) – это стандарт для обмена сообщениями между веб-сайтами и браузерами. Спустя 20 лет он получил крайне необходимое обновление. HTTP/2, благодаря внедрению таких концепций, как поддержка потоков, server push, сжатие заголовков и приоритизация, позволяет получить преимущества в аспектах скорости, безопасности и эффективности.

Вы научитесь всему, что вам нужно знать для эффективного использования HTTP/2. Узнаете, как оптимизировать веб-производительность с помощью новых функций, таких как фреймы, мультиплексирование и push. Изучите реальные примеры по управлению потоками и зависимостям.

Книга представляет собой практическое руководство с готовыми советами и передовыми методами, которое обязательно поможет вам быстро освоиться в мире HTTP/2!

Издание предназначено для веб-разработчиков и администраторов веб-сайтов.

«Отличная книга про новый стандарт HTTP/2 для начинающих с подробными объяснениями и очень хорошими примерами».
Ален Куньо, STIB-MIVB

«Подробно описано наиболее существенное изменение этого протокола за два десятилетия. Полезное чтиво для веб-разработчиков».
Рональд Крэнстон, Sky UK

«Незаменимый ресурс для изучения всех нюансов HTTP/2 и того, как он повлияет на дальнейшую разработку».
Том Маккерни, «Прикладные информационные науки»

«Самое простое объяснение HTTP/2, которое я видел на сегодняшний день. Настоятельно рекомендую».
Эдвин Квок, Red Soldier

Рассматриваемые темы:

- информация о HTTP/2 для веб-разработчиков;
- советы касаемо обновлений и устранения неполадок;
- примеры из реальной практики и тематические исследования;
- информация о QUIC и HTTP/3.

Барри Поллард – профессионал с двадцатилетним опытом разработки, поддержки и настройки программного обеспечения и инфраструктуры.

ISBN 978-5-97060-920-0





HTTP/2 в действии

HTTP/2



в действии

Барри Поллард





Барри Поллард

HTTP/2 в действии

HTTP/2 in Action

BARRY POLLARD



Shelter Island

HTTP/2 в действии

БАРРИ ПОЛЛАРД



Москва, 2021

УДК 004.432
ББК 32.972.1
П26

Поллард Б.
П26 HTTP/2 в действии / пер. с анг. П. М. Бомбаковой. – М.: ДМК Пресс, 2021. – 424 с.: ил.

ISBN 978-5-97060-920-0

После изучения этой книги читатели приобретут четкое представление о том, что представляют собой протокол HTTP/2 и связанные с ним технологии.

В части I обсуждается предыстория появления протокола и объясняется, в чем состоят его преимущества перед HTTP/1.1. Рассматриваются способы переноса веб-сайта на HTTP/2; приводятся инструкции по установке таких популярных веб-сервисов, как Apache, nginx, IIS. Часть II описывает сам протокол и порядок установки HTTP/2-соединения, рассказывает об основном формате фреймов HTTP/2; отдельная глава посвящена push-серверу HTTP/2, который является новой частью протокола. Часть III содержит информацию о глубинных компонентах протокола, на которые не могут повлиять ни веб-разработчики, ни администраторы веб-серверов, и освещает различия между реализациями развертывания протокола HTTP/2. Наконец, в части IV обозначены перспективы развития протокола HTTP и способы его дальнейшего улучшения.

Издание адресовано веб-разработчикам, администраторам веб-сайтов и тем, кто интересуется интернет-технологиями, в частности оптимизацией веб-производительности. В книге приведены ссылки, которые пригодятся читателям для дальнейшего изучения темы.

УДК 004.432
ББК 32.972.1

ISBN 978-1-6172-9516-4 (анг.)
ISBN 978-5-97060-920-0 (рус.)

В память о Ронане Рафферти (Ronan Rafferty) (1977–2018) –
веб-разработчике и друге

Оглавление

Содержание

Предисловие

Я рано заинтересовался темой HTTP/2. Появление новой технологии было интригующим. Она обещала почти неограниченный прирост производительности, потенциально устраняя необходимость в некоторых запутанных обходных путях, которые вынуждены были использовать веб-разработчики. Однако на практике все оказалось немного сложнее. Я потратил некоторое время на то, чтобы выяснить, как применить ее в моем сервере Apache. А затем я изо всех сил пытался объяснить увиденные мной результаты влияния на производительность. Ситуацию осложняло отсутствие документации. Я сделал пару публикаций в своем блоге о том, как применять эту технологию, и эти публикации пользовались популярностью. Вместе с этим я начал принимать участие в некоторых проектах, связанных с HTTP/2, на веб-сервисе GitHub, а также просматривать соответствующие темы в системе Stack Overflow и помогать тем, у кого были проблемы, схожие с моими. Когда со мной связалось издательство Manning и предложило начать работу над книгой о HTTP/2, я воспользовался этой возможностью. Я не участвовал в разработке этой технологии, однако я чувствовал, что могу помочь многим веб-разработчикам, столкнувшимся с трудностями. Они, как и я, слышали об этой технологии, но не обладали достаточными знаниями, чтобы применить ее.

За те полтора года, которые я писал эту книгу, технология HTTP/2 стала широко востребованной, и сейчас ее используют в разработке все большего количества веб-сайтов. Некоторые проблемы развертывания решались по мере обновления программного обеспечения. Я надеюсь, что некоторые из проблем, описанных в этой книге, со временем уйдут в прошлое. Я предполагаю, что нам потребуется еще несколько лет, чтобы полностью доработать HTTP/2.

Если вы научитесь использовать протокол HTTP/2, он обеспечит вам мгновенный прирост производительности веб-приложений без необходимости настройки или детального понимания. Однако в этой жизни ничего не дается даром, поэтому владельцам сайтов пойдет на пользу глубокое понимание всех тонкостей и нюансов протокола и его развертывания. Сегодня оптимизация производительности веб-приложений

находится в центре внимания, и HTTP/2 является еще одним инструментом, который даст нам новые интересные методики и возможности как сейчас, так и в будущем.

В сети доступно огромное количество информации для тех, у кого есть время и желание искать, фильтровать и понимать ее. Весьма приятно читать различные мнения и даже общаться непосредственно с разработчиками и исполнителями протоколов. Однако тема HTTP/2 очень обширна, и только объем и глубина книжного формата дают мне возможность полностью объяснить технологию, затрагивая связанные с ней темы, а также дать вам ссылки для дальнейшего изучения, если что-то вызовет у вас интерес. Я надеюсь, что в этой книге достиг своей цели.

Благодарности

Прежде всего я хотел бы поблагодарить мою невероятно понимающую жену Айне (Aine), которая провела последние полтора года, заботясь о наших двух маленьких детях (ставших «тремя маленькими детьми» во время написания этой книги!), в то время как я запирался в кабинете и яростно стучал по клавиатуре. Возможно, она была еще счастливее чем я, когда узнала, что эту книгу наконец опубликовали! Особую благодарность стоит выразить также моим родственникам (семье Бакли) (the Buckleys), которые помогали Айне развлекать наших детей и держать их подальше от моего домашнего кабинета, чтобы я смог сосредоточиться.

Команда издательства Manning оказывала мне огромную поддержку на протяжении всего процесса работы над книгой. В частности, я хотел бы поблагодарить Брайана Сойера (Brian Sawyer), который первым связался со мной и предложил мне написать эту книгу. Я ценю его помощь при подаче заявки на публикацию, благодаря чему книга была выбрана издателем. Кевин Харрелд (Kevin Harreld) проделал огромную работу в качестве консультирующего редактора, аккуратно подталкивая меня в правильном направлении и терпеливо отвечая на мои многочисленные вопросы. Томас Маккирни (Thomas McKearney) в роли технического редактора обеспечивал потрясающее техническое руководство, а также давал подробные отзывы по всем главам. Во время написания книги Иван Мартинович (Ivan Martinovic) организовал три этапа проверки. Они позволили получить обратную связь от руководства и узнать их мнение о том, что написано хорошо, а что нуждается в улучшении. Также хочу поблагодарить Матко Хрватина (Matko Hrvatin), который провел колоссальную работу по организации программы раннего доступа от Manning (MEAP), помогающей получить обратную связь от реальных читателей.

Еще я хотел бы поблагодарить всю маркетинговую группу издательства Manning, которая с самого начала помогала донести до читателей суть книги, а особую благодарность выразить Кристоферу Кауфману (Christopher Kaufman), который терпел мои, казалось, бесконечные просьбы о редактировании рекламного материала. Подготовка книги к выпуску была непростой задачей, поэтому спасибо Винсенту Нордхаусу (Vincent Nordhaus) за то, что он провел мой драгоценный результат через этот

процесс. Кэти Симпсон (Kathy Simpson) и Элисон Бренер (Alyson Brener) внесли неизмеримый вклад и провели редактирование и корректуру книги, сделав ее более читабельной. Перед ними стояла незавидная задача, так как я слишком часто задавал вопросы касаемо их поправок (гораздо лучших, чем моих!) относительно формулировки и грамматики. Спасибо также другим корректорам, дизайнерам, верстальщикам и наборщикам, которые работали на заключительных этапах написания этой книги. И хотя на обложке напечатано только мое имя, но все эти люди, среди прочих, помогли оформить мой извилистый ход мыслей в профессиональную публикацию. Любые присутствующие в книге ошибки – это, несомненно, только моя вина.

Я получил множество отзывов от людей за пределами издательства Manning, в частности от обозревателей книжного рынка и рецензентов рукописей (особенно спасибо тем из вас, кто принимал участие во всех трех обзорах!) до читателей MEAP. Отдельно я хотел бы поблагодарить Лукаса Пардью (Lucas Pardue) и Робина Маркса (Robin Marx), которые кропотливо изучали рукопись и предоставляли мне ценные экспертные знания о HTTP/2 на протяжении всего этого процесса. Среди других рецензентов хочу отметить такие имена, как Ален Кауньо (Alain Couniot), Анто Аравинт (Anto Aravinth), Арт Бергквист (Art Bergquis), Камаль Какар (Camal Caka), Дебмалья Джаш (Debmalya Jash), Эдвин Квок (Edwin Kwok), Итан Риветт (Ethan Rivett), Эван Уоллес (Evan Wallace), Флорин-Габриэль Барбучану (Florin-Gabriel Barbuceanu), Джон Мэтьюз (John Matthews), Джонатан Томс (Jonathan Thoms), Джошуа Хорвиц (Joshua Horwitz), Джастин Коулстон (Justin Coulston), Мэтт Деймел (Matt Deimel), Мэттью Фаруэлл (Matthew Farwell), Мэттью Халверсон (Matthew Halverson), Мортеза Киади (Morteza Kiadi), Рональд Крэнстон (Ronald Cranston), Райан Берроуз (Ryan Burrows), Сандип Хурана (Sandeep Khurana), Симеон Лейзерзон (Simeon Leyzerzon), Тайлер Коваллис (Tyler Kowallis) и Уэсли Бири (Wesley Beary). Спасибо вам всем.

Говоря о технологиях, я должен поблагодарить сэра Тима Бернерса-Ли (Sir Tim Berners-Lee) за то, что много лет назад он запустил всю эту веб-систему, а также Майка Белша (Mike Belshe) и Роберта Пеона (Robert Peon) за то, что вместе с Мартином Томпсоном (Martin Thompson), выступающим в качестве редактора, они изобрели SPDY, а затем оформили его как стандарт HTTP/2. Разработка и оформление стандарта стали возможными только благодаря трудолюбивым добровольцам рабочей группы инженеров Internet (Internet Engineering Task Force, IETF) и, в частности, рабочей группы HTTP, возглавляемой Марком Ноттингемом (Mark Nottingham) и Патриком МакМанусом (Patrick McManus). Без них всех, а также без разрешения их работодателей уделять время разработке не было бы HTTP/2 и, следовательно, не было бы необходимости в этой книге.

Я всегда удивлялся, как много времени и усилий технологическое сообщество вкладывает в волонтерскую работу. В процессе создания проектов с открытым исходным кодом, сайтов сообществ, таких как Stack Overflow, GitHub и Twitter, различных блогов и презентаций многие люди

тратят большую часть своего времени без какого-либо вознаграждения, оказывая помощь другим и расширяя свои собственные знания. Я благодарен тому, что являюсь частью этого сообщества, и очень горжусь этим. Эта книга не была бы написана без изучения трудов экспертов по веб-производительности Стива Саундерса (Steve Sounders), Йоава Вайса (Yoav Weis), Ильи Григорика (Ilya Grigorik), Пэта Мигана (Pat Meehan), Джейка Арчибальда (Jake Archibald), Хумана Бехешти (Hooman Beheshti) и Дэниела Стенберга (Daniel Stenberg), на которых я ссылался. Особую благодарность я выражаю Стефану Эйссингу (Stefan Eissing), проделавшему огромную работу над реализацией Apache HTTP/2, которая изначально привлекла мой интерес, и Тацухиро Цудзикаве (Tatsuhiro Tsujikawa), создавшему базовую библиотеку Nghttp2, входящую в упомянутую реализацию (наряду со многими другими разработками HTTP/2). Кроме того, написание этой книги стало возможным благодаря тому, что многие инструменты, такие как WebPagetest, HTTP Archive, W3Techs, Draw.io, TinyPng, nghttp2, curl, Apache, nginx и Let’s Encrypt в значительной части находятся в свободном доступе. Я хотел бы выразить особую благодарность тем компаниям, которые разрешили использовать свои инструменты при работе над этой книгой.

Наконец, я хотел бы поблагодарить вас, читатели, за проявленный интерес к этой книге. Хотя выпускать книгу мне помогали многие люди, так или иначе они делают это только ради таких людей, как вы, благодаря которым книги живут и становятся достойными публикации. Я надеюсь, что из этой книги вы извлечете ценные идеи и разберетесь в вопросах, касающихся HTTP/2.

Об этой книге

Я написал эту книгу для того, чтобы доходчиво и на реальных примерах объяснить читателю, как протокол работает на практике. Обычно спецификации протоколов изложены довольно сухо и сложны для понимания, поэтому цель данной книги – объяснить все детали на примерах, которые будут понятны каждому пользователю.

Кому следует прочесть эту книгу?

Эта книга создана для веб-разработчиков, администраторов веб-сайтов и тех, кто просто заинтересован в Internet-технологиях. Книга призвана полностью осветить тему HTTP/2 и все тонкости, связанные с протоколом. В блогах можно найти множество публикаций на эту тему, однако большинство из них написано для профессионалов или детально освещает лишь какой-либо отдельный аспект. Данная книга ориентирована на охват сразу всех аспектов функционирования протокола. Она должна подготовить читателя и помочь ему разобраться в спецификации протокола. Также, если после прочтения этой книги читатель захочет изучить тему более детально, ему будет гораздо проще понять некоторые сложные публикации в блогах. HTTP/2 был создан прежде всего для повышения производительности, поэтому любой, кто интересуется оптимизацией веб-производительности, обязательно извлечет из этой книги полезную информацию и сделает для себя некоторые выводы. Кроме того, в книге содержится множество ссылок, которые могут пригодиться для дальнейшего изучения темы.

Как устроена эта книга

Книга состоит из 10 глав и разделена на 4 части.

В части I говорится о предыстории появления протокола, а также о том, почему необходимо перейти на HTTP/2 и как это осуществить:

- в главе 1 дается *предыстория, необходимая для полного понимания книги*. Такой подход дает возможность пользователям, обладающим лишь базовыми знаниями, хорошо разобраться в теме;

- в главе 2 мы рассмотрим *проблемы с HTTP/1.1 и причины, по которым необходим был переход на HTTP/2*;
- в главе 3 мы *обсудим способы переноса вашего веб-сайта на HTTP/2 и некоторые сложности, связанные с этим процессом.* Эта глава дополняется приложением, содержащим инструкции по установке таких популярных веб-сервисов, как Apache, nginx, и IIS.

Часть II содержит еще больше полезной информации. В ней мы рассмотрим непосредственно сам протокол и его значение для практик веб-разработки:

- в главе 4 дается *описание характеристик протокола HTTP/2, порядок установки HTTP/2-соединения, а также рассказывается об основном формате фреймов HTTP/2*;
- глава 5 посвящена *push-серверу HTTP/2*, который является новой частью протокола. Он позволяет владельцам веб-сайтов отправлять ресурсы до того, как их запросят браузеры;
- в главе 6 мы рассмотрим *практическое значение HTTP/2 для веб-разработки.*

Часть III содержит информацию о глубинных компонентах протокола, на которые не могут повлиять ни веб-разработчики, ни даже администраторы веб-серверов:

- в главе 7 говорится *о других моментах, касающихся спецификации HTTP/2, таких как состояние, управление потоком и механизм приоритетности. Кроме того, здесь мы рассмотрим различия между реализациями развертывания протокола HTTP/2*;
- в главе 8 подробно рассматривается *протокол HPACK*, созданный для сжатия заголовков HTTP в HTTP/2.

В части IV мы поговорим о перспективах дальнейшего развития протокола HTTP:

- глава 9 посвящена *TCP, QUIC и HTTP/3.* Технологии постоянно развиваются, и теперь, когда протокол HTTP/2 стал доступен, разработчики уже ищут способы его улучшить. В этой главе мы обсудим недоработки в протоколе HTTP/2 и способы их устранения в его преемнике – HTTP/3;
- в главе 10 *рассмотрим другие способы улучшения HTTP помимо протокола HTTP/3*, а также поразмышляем о проблемах, которые были обнаружены во время разработки и оформления стандарта HTTP/2, и о том, актуальны ли эти проблемы в реальной практике.

После изучения этой книги читатели приобретут четкое представление о том, что представляет из себя протокол HTTP/2 и связанные с ним технологии. Также книга поможет лучше разобраться в оптимизации веб-производительности. Кроме того, когда QUIC и HTTP/3 станут доступны широкой публике, люди, познакомившиеся с этой книгой, будут готовы к работе с ними.

Примеры кода

Эта книга отличается от других книг технической направленности. Здесь вы не увидите большого количества кода, потому что книга посвящена именно протоколу, а не языку программирования. Книга нацелена на то, чтобы познакомить вас с методиками высокого уровня, которые вы сможете применить к любому веб-серверу или языку программирования, используемому для обслуживания страниц в сети. Однако в книге есть несколько примеров на языках NodeJS и Perl, а также фрагменты конфигурации веб-сервера.

Для того чтобы отделить программный код и фрагменты конфигурации от обычного текста, в книге мы используем моноширинный шрифт. В некоторых местах мы выделяем части кода жирным шрифтом, чтобы показать изменения, произошедшие в результате предыдущих шагов, описанных в главе, например когда к строке кода добавляется новая порция информации.

Все программные коды, используемые в примерах, доступны для загрузки с веб-сайтов издательства по адресу https://www.manning.com/books/http2-in-action или GitHub по адресу https://github.com/bazzadp/http2-in-action.

Онлайн-ресурсы

Остались вопросы?

- Официальная страница HTTP/2 доступна по адресу https://http2.github.io/. На этой странице вы сможете найти информацию о спецификации HTTP/2 и HPACK, реализации HTTP/2 и ответы на часто задаваемые вопросы.
- Официальная веб-страница рабочей группы HTTP доступна по адресу https://httpwg.org/. Большая часть информации о работе группы находится в открытом доступе на веб-странице GitHub https://github.com/httpwg/ и в списке рассылок (https://lists.w3.org/Archives/Public/ietf-http-wg/).
- Вы можете найти дополнительную информацию по тегу HTTP/2 в Stack Overflow: https://stackoverflow.com/questions/tagged/http2. Здесь же имеются ответы на вопросы от автора.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Барри Поллард (Barry Pollard) – профессиональный разработчик программного обеспечения. Почти 20 лет он занимается вопросами разработки и поддержки программного обеспечения и IT-инфраструктуры. Он проявляет большой интерес к веб-технологиям, повышению производительности, безопасности и практическому использованию технологий. Вы можете найти блог Барри Полларда на сайте https://www.tunetheweb.com. Кроме того, он ведет Твиттер (@tunetheweb).

Об иллюстрации на обложке

Иллюстрация на обложке книги называется «Одеяние русской торговки, 1768». Мы взяли ее из книги Томаса Джеффериса (Thomas Jefferys) *A Collection of the Dresses of Different Nations, Ancient and Modern*, опубликованной в Лондоне в 1757–1772 гг. На титульном листе указано, что это гравюра, выполненная вручную на медной пластине с использованием гуммиарабика. Томаса Джеффериса (1719–1771) называли «географом короля Георга III». Он был ведущим английским картографом того времени. Он гравировал и печатал карты для правительственных и других официальных органов, а также выпускал большое количество карт и атласов для обычных граждан; особенно он известен своими картами Северной Америки. Будучи картографом, он проявлял интерес к традиционной одежде, что носили люди, жившие на территории, которую он исследовал и наносил на карты. Эти наряды блестяще представлены в его коллекции. Он увлекался дальними странами и темой путешествий и делал это ради удовольствия. В конце XVIII века такой род занятий был относительно новым, и коллекции, подобные этой, были популярны, так как они знакомили туристов и людей, которые никогда не путешествовали, с культурой жителей других стран. Разнообразие иллюстраций в томах Джеффериса говорит нам о том, насколько уникальны и индивидуальны были культуры народов мира около 200 лет назад. С тех пор манера ношения одежды изменилась, а ее разнообразие в регионах и странах, столь богатое в то время, исчезло. Сегодня зачастую бывает трудно отличить жителей одного континента от жителей другого. Если взглянуть на это более оптимистично, возможно, мы предпочли культурному и визуальному разнообразию более насыщенную личную жизнь – или выбрали вариант разнообразить свою жизнь за счет интеллектуального развития или новых технологий. В наше время существует огромное количество компьютерной литературы, среди которой бывает трудно отличить одну книгу от другой. Издательство Manning особенно ценит изобретательность и инициативу в компьютерном бизнесе и старается отразить это в дизайне книжных обложек. Издательство выбрало именно такую обложку, потому что в этой иллюстрации Джефферис возвратил к жизни богатое разнообразие культур регионов двухвековой давности.

Часть I

Переход на HTTP/2

Чтобы понять, почему в области оптимизации производительности веб-приложений существует такой ажиотаж вокруг протокола HTTP/2, сначала вам необходимо узнать, зачем он нужен и какие проблемы он решает. Поэтому первая часть этой книги знакомит с HTTP/1 читателей, которые не знают, что это такое и как это работает. Затем мы объясним, почему было необходимо создание версии 2. Сначала поговорим о том, как работает HTTP/2 на более высоком уровне, а разбор работы нижних уровней оставим на потом. Вместо этого в конце первой части мы расскажем о различных методах, которые вы можете использовать для развертывания HTTP/2 на своем сайте.

1 Веб-технологии и HTTP

В этой главе мы рассмотрим:

- как браузер загружает веб-страницу;
- что такое HTTP и как появился HTTP/1.1;
- основы HTTPS;
- основные инструменты HTTP.

В этой главе говорится о том, как устроен веб в современном мире, а также дается объяснение некоторых ключевых концепций, которые помогут вам лучше понять смысл последующих глав книги. Затем мы поговорим о самом понятии HTTP и об истории версий протоколов, предшествующих ему. Мы полагаем, что многие читатели хотя бы немного знакомы с большей частью информации, заключенной в этой главе. Однако мы все же рекомендуем прочесть ее, чтобы освежить свои базовые знания.

ПРИМЕЧАНИЕ РЕДАКТОРА ПЕРЕВОДА Автор книги обращает наше внимание на различие между Всемирной паутиной (World Wide Web), представляющей собой вместилище общедоступного контента – сайтов, файлов, потоковых ресурсов и т. д., и сетью передачи данных Internet, которая представляет собой маршрутизируемые каналы передачи данных и набор протоколов разного уровня. Поскольку в этой книге речь идет именно о втором случае, мы, как и автор, далее используем термин «Internet» в его техническом понимании, а интернетом называем только Всемирную паутину.

1.1 О том, как работает сеть

Internet стал неотъемлемой частью повседневной жизни. В нем мы совершаем покупки, производим банковские операции, а также общаемся и развлекаемся. По мере развития *интернета вещей* (IoT – Internet of Things) к сети подключается все больше и больше устройств, что обеспечивает нам возможность получения удаленного доступа к ним. Реализация такого доступа стала возможной благодаря разработке нескольких технологий, в числе которых протокол передачи гипертекста (Hypertext Transfer Protocol, HTTP). Данный протокол является ключевым методом запроса удаленного доступа к веб-приложениям и ресурсам. Большинство людей умеет выходить в Internet через веб-браузер. Однако не каждый знает, как работает эта технология на самом деле. Многие люди не имеют понятия о том, почему HTTP является основной частью веба и почему следующая версия (HTTP/2) вызывает такой ажиотаж в веб-сообществе.

1.1.1 Internet и Всемирная паутина

Многие люди считают, что понятия Internet и «Всемирная паутина» (или просто «веб») являются синонимами, но на самом деле это два разных термина, которые очень важно различать.

Internet представляет собой глобальную структуру объединенных компьютерных сетей, использующих общий интернет-протокол (Internet Protocol, IP) для маршрутизации сообщений. Он состоит из множества сервисов, таких как веб, электронная почта, система предоставления общего доступа к файлам и интернет-телефония. Таким образом, веб является лишь одной из систем, существующих в сети Internet, хотя и самой заметной. Люди часто пользуются электронной почтой с помощью внешних веб-интерфейсов (например, Gmail, Hotmail и Yahoo!) и поэтому ошибочно отождествляют понятия «веб» и Internet.

HTTP – это протокол, при помощи которого веб-браузер запрашивает и получает веб-страницы с сервера. Это одна из трех основных технологий, созданных Тимом Бернерсом-Ли (Tim Berners-Lee), который также является автором Всемирной паутины (worldwide web, WWW), уникальных идентификаторов ресурсов (ставших основой для унифицированных указателей ресурсов или URL-адресов) и языка гипертекстовой разметки (Hypertext Markup Language, HTML). Другие системы в Internet созданы по своим собственным протоколам и стандартам, определяющим их работу, а также маршрутизацию их сообщений (например, маршрутизация электронной почты происходит посредством протоколов SMTP, IMAP и POP). HTTP связан именно с вебом, однако эта граница постепенно размывается. Появление сервисов без традиционных клиентских веб-интерфейсов, основанных на HTTP, означает, что определить границы Всемирной паутины становится все сложнее и сложнее! Такие сервисы (известные под аббревиатурами REST и SOAP) могут быть применимы

как к веб-страницам, так и к другим ресурсам, размещенным в Internet (например, к мобильным приложениям). Интернет вещей представляет собой совокупность устройств, предоставляющих пользователям определенные функции. С ними могут взаимодействовать другие устройства (компьютеры, мобильные приложения и даже другие устройства интернета вещей). В большинстве случаев связь между ними происходит посредством HTTP-запросов. Таким образом, посредством отправки HTTP-сообщения вы можете включить или выключить лампочку, например, с помощью приложения на своем мобильном телефоне.

Хотя в структуре Internet существует огромное количество сервисов, большая их часть со временем используется все реже и реже, в то время как количество пользователей Всемирной сети продолжает расти. Некоторые читатели вспомнят такие аббревиатуры, как BBS и IRC. Сегодня они считаются устаревшими, а на их место пришли веб-форумы, социальные сети и мессенджеры.

Как уже было сказано, термин «Всемирная паутина» часто ошибочно отождествляется с понятием Internet. Однако непрерывное развитие Всемирной сети (а также специально созданного для нее протокола HTTP) может означать, что очень скоро такое понимание окажется не столь далеким от истины, как раньше.

1.1.2 Что происходит, когда вы просматриваете веб-страницы?

Сейчас мы возвращаемся к первоначальному (и основному) назначению HTTP: передаче веб-запросов. Когда вы открываете веб-сайт в своем любимом браузере на стационарном или портативном компьютере, планшете, мобильном телефоне или на любом другом из множества устройств, обеспечивающих доступ в Internet, происходит очень много процессов. Чтобы извлечь максимум пользы из этой книги, вам необходимо понять, какие процессы происходят, когда мы загружаем веб-страницу.

Предположим, что вы запускаете браузер и переходите на сайт www.google.com. В течение нескольких секунд произойдет процесс, показанный на рис. 1.1.

1. Браузер запросит реальный адрес www.google.com с сервера системы доменных имен (Domain Name System, DNS), который переведет понятное человеку имя www.google.com в удобный для машины IP-адрес.

   Если представить, что IP-адрес – это телефонный номер, то DNS – это телефонная книга. Этот IP-адрес будет являться либо адресом более старого формата IPv4, который относительно понятен человеку (например, 216.58.192.4), либо адресом нового формата IPv6, который могут обрабатывать только машины (например, 2607:f8b0:4005:801:0:0:0:2004). Когда в городе заканчиваются свободные телефонные номера, приходится менять телефонный код города. Приблизительно по этой же причине введен протокол IPv6. Он не-

обходим, чтобы справиться с ростом количества устройств, подключающихся к Internet сейчас и в будущем.

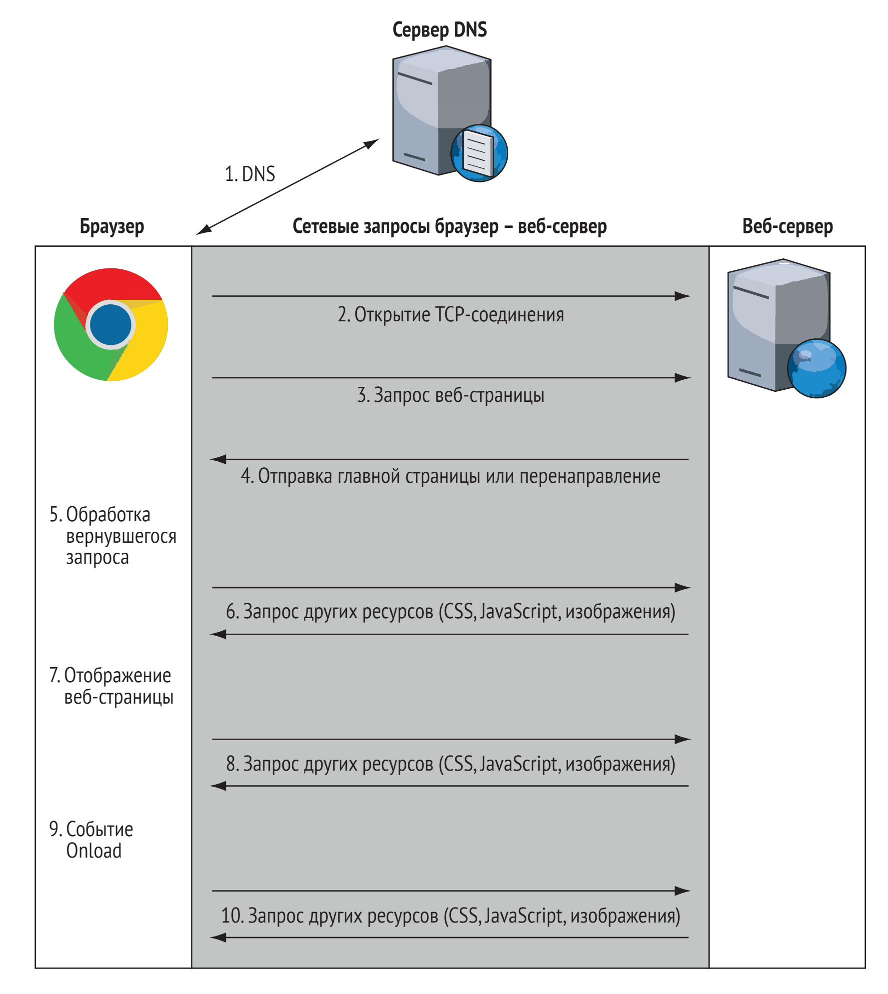


Рис. 1.1 Типичное взаимодействие при просмотре веб-страницы

Имейте в виду, что по причине глобального характера сети Internet крупные компании часто имеют несколько серверов по всему миру. Когда вы запрашиваете IP-адрес у DNS, он обычно предоставляет IP-адрес ближайшего сервера, благодаря чему вы получаете наиболее быстрый доступ к веб-страницам. Например, ответ на запрос IP-адреса для www.google.com в Америке и в Европе будет отличаться. Поэтому не волнуйтесь, если ваши значения IP-адресов для www.google.com будут отличаться от тех, что вы найдете в нашей книге.

Где же версия IPv5?

Если Internet-протокол версии 4 (IPv4) был заменен версией 6 (IPv6), то где же тогда версия 5? И почему вы никогда не слышали об IPv1 или IPv3?

Первые четыре бита в IP-пакете используются для хранения версии протокола. В теории максимально возможная версия – это версия 15. До того как IPv4 начали использовать повсеместно, существовало четыре экспериментальных версии (версия 0 – версия 3). Однако ни для одной из них, кроме версии 4, не было создано официального стандарта[a]. Затем создали версию 5, которая предназначалась для протокола реального времени Internet Stream Protocol, обеспечивающего потоковую передачу аудио и видео в реальном времени, подобно современной IP-телефонии (Voice over IP, VoIP). Однако версия IPv5 так и не стала популярной, так как имела те же ограничения адресного пространства, что и версия 4. Когда появилась версия 6, работа над IPv5 была остановлена, и версия 6 стала преемником IPv4. Судя по всему, изначально IPv6 была названа версией 7, потому что многие ошибочно полагали, что версия 6 уже использована[b]. Номера 7, 8 и 9 также применялись для нумерации версий, но эти версии больше не используются. Если когда-нибудь и появится преемник IPv6, то это, скорее всего, будет IPv10 или более поздняя версия. В связи с этим, несомненно, возникнут вопросы, подобные тем, с которых начинается это примечание!

[a] Стандарт протокола https://tools.ietf.org/html/rfc760 позже был обновлен и заменен на https://tools.ietf.org/html/rfc791.

[b] Обратите внимание на информацию в публикации https://archive.is/QqU73#selection-417.1-417.15.

2 Через веб-браузер ваш компьютер устанавливает TCP-соединение[1] по IP-адресу через стандартный сетевой порт (порт 80)[2] или стандартный защищенный сетевой порт (порт 443).

Передача информационных потоков в Internet осуществляется с помощью протокола IP. Протокол TCP обеспечивает надежность при передаче данных, а также возможность повторной передачи («Здравствуйте, вы все поняли?», «Нет, не могли бы вы повторить последний бит, пожалуйста?»). Эти две технологии часто используются вместе, поэтому их называют TCP/IP. Именно на основе этих протоколов создана большая часть сервисов в Internet.

Один сервер может использоваться несколькими службами (например, электронная почта, FTP, HTTP и HTTPS [HTTP Secure]-серверы). Порт позволяет различным службам размещаться под

[1] Google начал экспериментировать с протоколом QUIC, поэтому, если вы подключаетесь к сайту Google из Chrome, то может быть задействована эта технология. Я расскажу про QUIC в главе 9.

[2] Некоторые веб-сайты, в том числе Google, используют технологию HSTS для автоматической активации защищенного HTTP-соединения (HTTPS) через порт 443, поэтому, даже если вы попытаетесь подключиться через HTTP, соединение автоматически обновится до HTTPS до отправки запроса.

одним IP-адресом, так же как, например, в компании может быть добавочный номер телефона для каждого сотрудника.

3 После того как браузер подключился к веб-серверу, он начинает отправлять запросы веб-сайту. На этом этапе в дело вступает протокол HTTP. Однако мы рассмотрим порядок его работы в следующем разделе. А сейчас сосредоточимся на работе браузера и рассмотрим пример, где с помощью HTTP он запрашивает у сервера Google домашнюю страницу.

ПРИМЕЧАНИЕ На этом этапе ваш браузер автоматически исправляет сокращенный адрес (www.google.com) на синтаксически верный URL-адрес (http://www.google.com). В абсолютном полном URL-адресе указывается номер порта (например, http://www.google.com:80). Если используются стандартные порты (80 для HTTP и 443 для HTTPS), то веб-браузер скроет их номера. Если используются нестандартные порты, их номера будут отображаться в URL. Например, некоторые системы, особенно в среде разработки, используют порт 8080 для HTTP или 8443 для HTTPS.

Если используется протокол HTTPS (более подробно мы рассмотрим его в разделе 1.4), то для настройки шифрования, защищающего соединение, требуются дополнительные шаги.

4 Сервер Google отвечает вне зависимости от типа URL-адреса, который вы запрашиваете. Как правило, ответ приходит в виде текста веб-страницы в формате HTML. HTML – *это стандартизированная, структурированная система разметки текстового содержимого веб-страниц*. Документ на языке HTML обычно представляет собой набор элементов, начало и конец которых определяется HTML-тегами и ссылками на блоки другой информации, необходимой для создания мультимедийных веб-страниц, которые вы привыкли видеть (каскадные таблицы стилей [CSS], код JavaScript, изображения, шрифты и т. д.).

Однако иногда вместо HTML-страницы ответом может быть перенаправление на верный адрес. Например, Google работает только на HTTPS, поэтому, если вы запросите URL http://www.google.com, ответом будет специальная HTTP-инструкция (обычно код такого ответа `301` или `302`), которая перенаправит вас на новый адрес https://www.google.com. Такой ответ запускает процессы некоторых или всех предыдущих шагов снова, в зависимости от того, как выглядит адрес перенаправления. Он может иметь другой порт на другом сервере, другой порт на том же сервере (например, перенаправление на HTTPS) или это может быть другая страница на том же сервере и порте.

При возникновении ошибки вы увидите на экране ответ в виде HTTP-кода, самым известным из которых является `404 Not Found`.

5 Затем браузер обрабатывает возвращенный запрос. Браузер рассчитан на ответы в формате HTML, поэтому он начинает анализи-

ровать HTML-код и строит в памяти объектную модель документа (document object model, DOM), которая отображает внутреннее строение страницы. Скорее всего, в процессе анализа веб-браузер найдет и другие ресурсы, необходимые для правильного отображения страницы (например, CSS, JavaScript и изображения).

6 Веб-браузер запросит необходимые ему дополнительные ресурсы. Веб-страницы Google используют небольшое количество ресурсов. *На момент написания этой книги их требуется всего 16*. Каждый из этих ресурсов запрашивается аналогичным образом, следуя шагам 1–6, так как эти ресурсы в свою очередь могут запрашивать другие ресурсы. Как правило, страницы обычных веб-сайтов более разнообразны, чем страницы Google. Такие веб-сайты нуждаются в 75 ресурсах[1] и зачастую из разных доменов, поэтому шаги 1–6 должны быть выполнены для каждого ресурса. Именно из-за таких ситуаций и замедляется работа браузера. Протокол HTTP/2 помогает оптимизировать запрос дополнительных ресурсов. Об этом мы поговорим в следующих главах.

7 Когда браузер получает достаточно критически необходимых ресурсов, он начинает отображать страницу на экране. Старт рендеринга веб-страницы является не таким простым процессом, как кажется. Браузеру требуется много времени для загрузки ресурсов, необходимых для отображения веб-страницы. Поэтому соединение становится еще более медленным и не дает ожидаемых пользователем результатов. Если веб-браузер начнет отображать страницу слишком рано, она будет «прыгать» по мере загрузки большего количества контента. Особенно раздражает, когда страница начинает «прыгать» после того, как вы прочитали уже половину статьи. Четкое понимание технологий, на которых основан веб, особенно HTTP и HTML/CSS/JavaScript, помогает владельцам веб-сайтов убрать эти раздражающие скачки и оптимизировать загрузку страниц, однако многие из них все еще этого не делают.

8 После отображения страницы веб-браузер продолжает в фоновом режиме загружать другие ресурсы, необходимые странице, и обновляет страницу по мере их обработки. Он загружает второстепенные элементы, такие как изображения и скрипты отслеживания рекламы. Именно поэтому бывает так, что, когда вы загружаете веб-страницу, на ней какое-то время не отображаются изображения (особенно при медленном соединении), а затем, по мере их загрузки, они появляются.

9 Когда страница полностью загружена, браузер останавливает значок загрузки (в большинстве браузеров вращающийся значок в адресной строке или рядом с ней) и запускает событие `OnLoad` JavaScript, которое является маркером для JavaScript и означает, что страница готова к работе.

---

1 https://httparchive.org/reports/page-weight#reqTotal.

10 Времена, когда веб-страница была статична, уже давно прошли, поэтому браузер продолжает отправлять запросы, даже когда она полностью загружена. Сегодня многие веб-страницы являются многофункциональными приложениями, взаимодействующими с различными серверами в Internet для отправки или загрузки дополнительного контента. Отображение контента может быть инициировано действиями пользователя. Например, вы вводите запросы в строку поиска на домашней странице Google и мгновенно видите предложенные варианты запроса без нажатия кнопки поиска. Также это могут быть действия, управляемые приложением, например автоматическое обновление ленты Facebook или Twitter без нажатия кнопки обновления. Эти действия часто происходят в фоновом режиме. Они скрыты от вас. В качестве примера можно привести рекламные и аналитические скрипты, которые отслеживают ваши действия на сайте, для того чтобы сообщать аналитические данные владельцам веб-сайтов и/или рекламным сетям.

Как видите, после того как вы вводите URL-адрес, происходит огромное количество процессов, и в большинстве случаев они протекают очень быстро. Описание любого из этих шагов могло бы лечь в основу целой книги с различными вариациями при определенных обстоятельствах. В нашей книге мы делаем упор на шаги 3–8 (загрузка веб-сайта с помощью HTTP). Также в некоторых главах (в частности, в главе 9) рассказывается о шаге 2 (базовое сетевое соединение, используемое HTTP).

1.2 Что такое HTTP?

Предыдущий раздел освещает детали протокола HTTP, благодаря чему вы имеете представление о его функционировании в контексте всего Internet. В этом разделе мы кратко опишем, как данный протокол функционирует и где он используется.

Как уже было сказано, *HTTP расшифровывается как «протокол передачи гипертекста»*. Как следует из его названия, HTTP изначально предназначался исключительно для передачи гипертекстовых документов (документов, содержащих ссылки на другие документы). Первая версия не могла передать ничего, кроме таких документов. Название «протокол передачи гипертекста» не совсем актуально, так как разработчики быстро поняли, что протокол может быть использован для передачи других типов файлов (например, изображений). Однако, сейчас HTTP настолько распространен, что переименовывать его уже слишком поздно.

Сетевая модель TCP/IP, построенная на одном из типов физического соединения (Ethernet, Wi-FI и т. д.), обеспечивает надежное сетевое соединение, необходимое для функционирования HTTP. Структура протоколов передачи данных разделена на уровни, каждый из которых отвечает за определенный процесс. HTTP не имеет отношения к установке сетевого соединения. Несмотря на то что HTTP-приложения должны

уметь справляться с сетевыми сбоями или отключениями, сам протокол не предполагает решения этих задач.

Модель взаимодействия открытых систем (Open Systems Interconnection, OSI) – *это многоуровневая модель сетевых протоколов*. Она состоит из 7 уровней, хотя они соотносятся с сетями, и особенно с Internet-трафиком, с некоторой долей условности. TCP (Transmission Control Protocol) является рабочим протоколом как минимум двух уровней, а возможно, и трех (это зависит от того, как вы определяете эти уровни). На рис. 1.2 вы можете увидеть, как данная модель соотносится с Internet-трафиком и как в нее вписывается протокол HTTP.

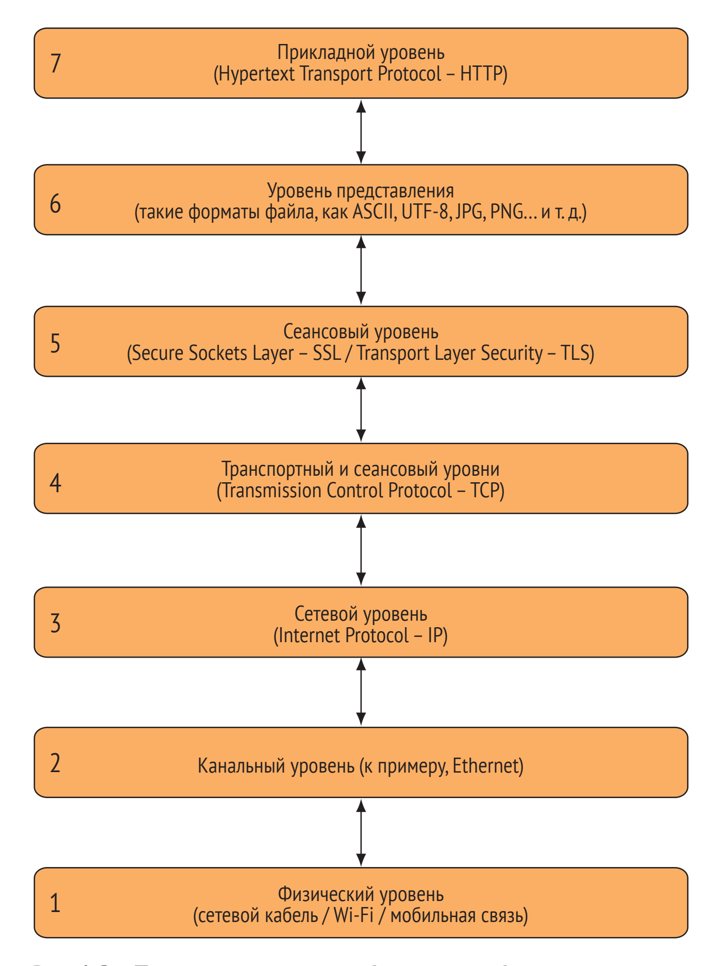


Рис. 1.2 **Транспортные уровни Internet-трафика**

К слову, сегодня до сих пор ведутся споры о точном определении уровней в модели. В Internet, как и в других сложных системах, не все поддается классификации, и разграничить что-либо бывает не так легко, как этого хотелось бы разработчикам. На самом деле, согласно мнению Ин-

женерного совета интернета, не обязательно делать слишком большой упор на деление модели по уровням[1]. Однако на более высоком уровне такой подход может помочь понять, как в модель вписывается HTTP, а также как она зависит от других протоколов. Многие веб-приложения используют HTTP, поэтому в таких случаях прикладной уровень может больше относиться к сетевому уровню, чем к приложениям JavaScript.

По сути, HTTP является протоколом запроса и ответа. Браузер совершает запрос к веб-серверу, используя синтаксис HTTP. Веб-сервер в свою очередь отвечает сообщением, содержащим запрошенный ресурс. HTTP широко распространен именно благодаря своей простоте. Однако, как вы увидите в последующих главах, HTTP/2 немного сложнее, чем HTTP, но в то же время он и более эффективен.

После того как открывается соединение, базовый синтаксис HTTP-запроса выглядит следующим образом:

```
GET /page.html↵,
```

где символ ↵ означает возврат каретки / начало новой строки (клавиша **Enter** или **Return**). HTTP очень прост в своей базовой форме! Вы указываете один из нескольких HTTP-методов (в данном случае `GET`), за которым следует нужный вам ресурс (`/page.html`). Помните, что на этом этапе вы уже подключились к соответствующему серверу посредством TCP/IP, поэтому просто запрашиваете нужный ресурс с этого сервера. Вас не должно беспокоить то, как работает и чем управляется это соединение.

Первая версия HTTP (0.9) поддерживала только базовый синтаксис и могла использовать только метод `GET`. В данном случае вам может быть интересно, зачем использовали метод `GET` для запроса HTTP 0/9. Здесь он кажется неуместным, но этот и другие методы появились в последующих версиях HTTP, так что спасибо изобретателям HTTP за то, что они предвидели их появление. В следующем разделе мы обсудим различные версии протокола HTTP, однако синтаксис по-прежнему соответствует формату запроса HTTP `GET`.

Рассмотрим пример из реальной жизни. Для получения HTTP-запросов веб-серверу требуется только соединение TCP/IP, и вы можете эмулировать работу браузера с помощью протокола Telnet. *Это простой протокол, который реализует соединение с сервером с помощью TCP/IP и позволяет вводить текстовые команды, а также просматривать текстовые ответы*. Данный протокол необходим при использовании HTTP, но ближе к концу главы мы рассмотрим и другие, более подходящие инструменты для просмотра HTTP-страниц. К сожалению, некоторые технологии теряют свою популярность, и Telnet является одной из них; многие операционные системы больше не включают в себя клиент Telnet по умолчанию. Возможно, вам потребуется установить клиент Telnet, чтобы попробовать реализовать некоторые простые команды HTTP, или же вы можете использовать в качестве эквивалента утилиту nc (netcat), которая уже установлена в большинстве Linux-подобных систем, вклю-

---

[1] https://tools.ietf.org/html/rfc3439#section-3.

чая macOS. Для рассмотрения простых примеров, которые представлены в нашей книге, она подойдет почти также, как и Telnet.

Для Windows лучше использовать утилиту PuTTY[1] вместо клиента, поставляемого в комплекте с Windows по умолчанию (который обычно не устанавливается и должен добавляться вручную). Клиент, установленный по умолчанию, имеет проблемы с отображением: например, он может не отображать то, что вы печатаете, или записывать новую информацию поверх уже имеющейся. После установки и запуска PuTTY вы увидите окно конфигурации, в котором вы можете ввести хост (www.google.com), порт (80) и тип подключения (Telnet). Убедитесь, что вы выбрали опцию **Never** (Никогда) для закрытия окна после окончания сеанса; в противном случае вы не увидите результатов. Все эти настройки вы можете наблюдать на рис. 1.3. Также обратите внимание на то, что вы можете изменить режим Telnet Negotiation на пассивный (**Connections** > **Telnet** > **Telnet Negotiation**) в случае, если у вас возникли проблемы с вводом любой из следующих команд и вы получаете сообщения о неправильном формате запроса.

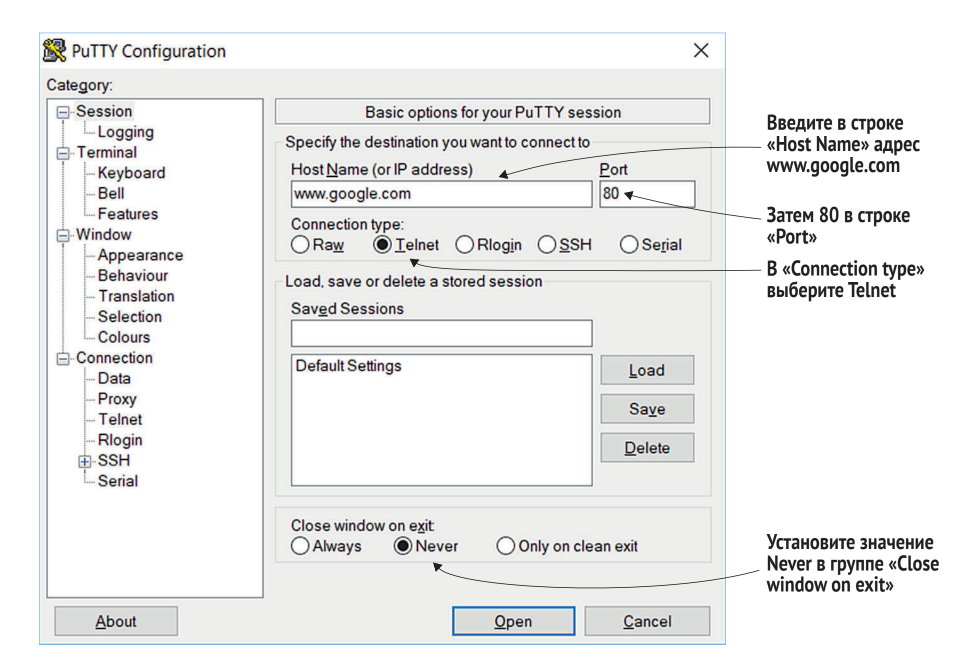


Рис. 1.3 Настройки PuTTY для соединения с Google

Если вы используете компьютер Apple Macintosh или ОС Linux, вы можете выполнить команду Telnet непосредственно из командной строки оболочки, если Telnet уже установлен:

```
$ telnet www.google.com 80
```

---

[1] https://www.putty.org/.

или, как я уже упоминал ранее, используйте команду `nc` таким же образом:

```
$ nc www.google.com 80
```

Когда вы открываете сеанс Telnet и устанавливаете соединение, вы видите пустой экран или, в зависимости от вашего приложения Telnet, некоторые команды, такие как:

```
Trying 216.58.193.68...
Connected to www.google.com.
Escape character is '^]'.
```

Независимо от того, отображается это сообщение или нет, вы должны иметь возможность вводить свои HTTP-команды, поэтому введите `GET/`, а затем нажмите клавишу ввода, которая сообщает серверу Google, что вы ищете страницу по умолчанию (/) и (поскольку вы не указали версию HTTP) хотите использовать HTTP/0.9. Обратите внимание, что некоторые клиенты Telnet не повторяют то, что вы печатаете по умолчанию (особенно клиент Telnet по умолчанию в комплекте Windows, как я уже упоминал ранее), поэтому может быть трудно точно увидеть, что вы печатаете. Но вы все равно должны отправлять команды.

Использование Telnet через прокси-сервер организации

Если у вашего компьютера нет прямого доступа в Internet, вы не сможете подключиться к Google напрямую с помощью Telnet. Этот сценарий часто встречается в корпоративных сетях, где для ограничения прямого доступа используется прокси (мы поговорим о прокси-серверах в главе 3). В этом случае вы можете использовать в качестве примера один из ваших внутренних веб-серверов (например, ваш сайт во внутренней сети), а не Google. В разделе 1.5.3 мы рассмотрим другие инструменты, которые могут работать с прокси, но пока стоит сосредоточиться на Telnet.

Скорее всего, сервер Google ответит, используя HTTP/1.0, несмотря на то, что вы запросили HTTP/0.9 по умолчанию (так как ни один сервер больше не использует HTTP/0.9). В ответ появится код ответа `200` (если команда была выполнена) или `302` (если сервер хочет перенаправить вас в другое место). Затем соединение закроется. Более подробно я расскажу об этом процессе в следующем разделе, поэтому не стоит сейчас зацикливаться на этих деталях.

Ниже приведен один из некоторых ответов из командной строки на сервере Linux, выделенный жирным шрифтом. Стоит отметить, что возвращаемый HTML-контент не отображается полностью для краткости:

```
$ telnet www.google.com 80
Trying 172.217.3.196...
Connected to www.google.com.
Escape character is '^]'.
GET /
HTTP/1.0 200 OK
```

```
Date: Sun, 10 Sep 2017 16:20:09 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See
     https://www.google.com/support/accounts/answer/151657?hl=en for more info.
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie:
     NID=111=QIMb1TZHhHGXEPjUXqbHChZGCcVLFQOvmqjNcUIejUXqbHChZKtrF4Hf4x4DVjTb01R
     8DWShPlu6_aQ-AnPXgONzEoGOpapm_VOTW0Y8TWVpNap_1234567890-p2g; expires=Mon,
     12-Mar-2018 16:20:09 GMT; path=/; domain=.google.com; HttpOnly
Accept-Ranges: none
Vary: Accept-Encoding

<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"
     lang="en"><head><meta content="Search the world's information, including
     webpages, images, videos and more. Google has many special features to help
     you find exactly what you're looking for." name="description

…и т. д.

</script></div></body></html>Connection closed by foreign host.
```

Если вы находитесь за пределами Соединенных Штатов, вы можете увидеть перенаправление на ваш местный сайт Google вместо указанного здесь адреса. Например, если вы находитесь в Ирландии, Google отправляет ответ 302 и советует браузеру перейти на Google Ireland (http://www.google.ie), как показано здесь:

```
GET /
HTTP/1.0 302 Found
Location: http://www.google.ie/?gws_rd=cr&dcr=0&ei=BWe1WYrf123456qpIbwDg
Cache-Control: private
Content-Type: text/html; charset=UTF-8
P3P: CP="This is not a P3P policy! See
     https://www.google.com/support/accounts/answer/151657?hl=en for more info."
Date: Sun, 10 Sep 2017 16:23:33 GMT
Server: gws
Content-Length: 268
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: NID=111=ff1KAwIMjt3X4MEg_KzqR_9eAG78CWNGEFlDG0XIf7dLZsQeLerX
     P8uSnXYCWNGEFlDG0dsM-8V8X8ny4nbu2w96GRTZtzXWOHvWS123456dhd0LpD_123456789;
     expires=Mon, 12-Mar-2018 16:23:33 GMT; path=/; domain=.google.com; HttpOnly

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">

     <TITLE>302 Moved</TITLE></HEAD><BODY>  <H1>302 Moved</H1>
          The document has moved
               <A
HREF="http://www.google.ie/?gws_rd=cr&amp;dcr=0&amp;ei=BWe1WYrfIojUgAbqpIbwDg">he
re</A>.
</BODY></HTML> Connection closed by foreign host.
```

Как показано в конце каждого примера, после ответа сервера соединение закрывается. Следовательно, чтобы отправить другую команду HTTP, вам нужно заново открыть соединение. Чтобы избежать этого шага, вы можете использовать протокол HTTP/1.1 (который держит соединение открытым по умолчанию, но об этом я расскажу позже), введя `HTTP/1.1` после запрошенного ресурса:

```
GET / HTTP/1.1↵↵
```

Заметьте, что, если вы используете протокол HTTP/1.0 или HTTP/1.1, вы должны нажать клавишу ввода дважды, чтобы сообщить веб-серверу, что вы закончили отправку HTTP-запроса. В следующем разделе я расскажу, почему для соединений HTTP/1.0 и HTTP/1.1 требуется двойной перевод строки.

После того как сервер ответит, вы можете повторно выполнить команду `GET`, чтобы снова открыть страницу. На практике веб-браузеры обычно используют это открытое соединение для получения других ресурсов, а не того же самого ресурса снова, но концепция одинаковая.

Технически, чтобы соблюдать спецификацию HTTP/1.1, запросы HTTP/1.1 также требуют от вас указания заголовка хоста по причинам, которые мы (опять же) рассмотрим позже. Однако для этих простых примеров не стоит слишком беспокоиться об этом требовании, потому что Google, похоже, не настаивает на нем (хотя, если вы используете другие сайты, а не www.google.com, вы можете увидеть неожиданные результаты).

Как вы можете видеть, основной синтаксис HTTP прост. Это текстовый формат запроса–ответа, хотя он изменился в HTTP/2, где перешел в двоичный формат.

Если вы запрашиваете нетекстовые данные, такие как изображение, программы Telnet будет недостаточно. В терминальном сеансе появится бессмыслица, когда Telnet попытается, но не сможет преобразовать двоичный формат изображения в осмысленный текст, как в этом примере:

```
$ telnet www.google.com 80
Trying 172.217.3.164...
Connected to www.google.com.
Escape character is '^]'.
GET /images/branding/googlelogo/2x/googlelogo_color_120x44dp.png
▒.k▒
I▒¥&▒]▒S▒y▒8▒▒▒▒.κ?F▒{I▒iH▒g▒?Sk▒"▒▒F>#U▒p▒I▒7E^T▒~n▒EG▒I▒^▒
     +.`x▒\w ▒CR怛▒U3V▒O>6b▒y8▒S▒cHj^.▒▒F▒4=xw
(▒F▒Bc▒]Zu▒ ▒~Hj▒i▒▒R▒G▒mH▒.|▒▒ <▒▒xH걱▒
     ▒▒;▒'▒fH▒5▒ʃu▒▒|▒%WH▒7's/ÿ▒wʌ▒
▒b▒@▒4▒ـصـ▒{:$▒.▒(O▒▒ÿ▒ Ў▒=▒!i▒▒▒\▒DɣM▒▒9
     ▒
.▒$I▒$I▒$I▒$I▒▒▒~▒T▒LC▒
▒IEND▒B`▒Connection closed by foreign host.
```

В настоящее время Telnet уже не является актуальным инструментом для работы с HTTP-запросами, так как сегодня доступны более удобные варианты. Однако на примере Telnet легко объяснить сущность форма-

та HTTP-сообщения, а также показать то, насколько простыми были начальные версии протокола.

Как уже было сказано, HTTP обрел свою популярность благодаря простоте в его реализации и обслуживании. Таким образом, практически любой компьютер при наличии сетевых возможностей (от сложных серверов до лампочек из мира интернета вещей) может работать с HTTP и быстро передавать нужные команды по сети. Более сложной задачей является реализация веб-сервера, работающего исключительно с помощью HTTP. Работа браузеров также чрезвычайно сложна. После того как вы открыли веб-страницу с помощью HTTP, они начинают взаимодействовать со множеством других протоколов (сюда же входят HTML, CSS и JavaScript, благодаря которым страницы отображаются в веб-браузере). Однако создать простую программу, которая будет получать HTTP-запрос GET и давать ответ в виде данных, не так уж сложно. Кроме того, простота использования HTTP привела к буму в области микросервисов, где приложение разбивается на множество независимых веб-сервисов, зачастую основанных на более легких серверах приложений, таких как Node.js (Node).

1.3 Синтаксическая структура HTTP и история его создания

Создателем протокола HTTP является Тим Бернерс-Ли, работавший в то время со своей командой в исследовательской организации ЦЕРН (CERN) в 1989 году. Изначально HTTP был задуман как способ реализации сети взаимосвязанных компьютеров, целью создания которой являлась возможность обеспечить доступ к исследованиям и связать их. Предполагалось, что компьютеры в этой сети смогут легко ссылаться друг на друга в режиме реального времени (достаточно будет кликнуть по ссылке – и откроется связанный документ). Идея создания такой системы зародилась достаточно давно, а термин «гипертекст» появился еще в 1960-х. 1980-е характеризуются бурным ростом и развитием Internet. Неудивительно, что именно в то время и появилась возможность реализовать эту идею. В 1989 и 1990 годах Бернерс-Ли опубликовал предложение[1] о создании такой системы. Кроме того, он создал первый веб-сервер на основе HTTP и первый веб-браузер, который мог запрашивать HTML-документы и отображать их.

1.3.1 HTTP/0.9

В 1991 году была опубликована первая спецификация[2] протокола HTTP версии 0.9. Она изложена в краткой форме и состоит менее чем из 700 слов. Согласно этой спецификации при использовании HTTP 0.9 со-

1 https://www.w3.org/History/1989/proposal.html.
2 https://www.w3.org/Protocols/HTTP/AsImplemented.html.

единение с сервером и дополнительным портом (если порт не указан, то по умолчанию происходит соединение с портом 80) осуществляется по протоколу TCP/IP (или с помощью аналогичной службы, ориентированной на установку предварительного соединения). Для этого следует сделать запрос в виде всего одной строки ASCII-текста, состоящей из команды `GET`, адреса документа (без пробелов) и символов возврата каретки и перевода строки (возврат каретки необязателен). Сервер должен ответить сообщением в формате HTML, которое в спецификации описано как «байтовый поток символов ASCII». После каждого запроса соединение закрывается сервером (как это было показано в предыдущих примерах). Также, согласно спецификации, «ответы на ошибки предоставляются в виде понятного человеку текста в формате HTML». Отличить ответ об ошибке от корректного ответа можно только по содержимому текста, иного способа не существует. Ответ заканчивается так: «Запросы идемпотентны. Сервер не сохранит данные о запросе при отключении от сети». HTTP 0.9 не фиксирует данные о запросах, поэтому с одной стороны он является благословлением (так как он очень прост), а с другой – проклятием (так как для создания сложных приложений необходимо использовать другие технологии, например HTTP-куки). Ниже приведена единственная возможная команда для HTTP/0.9:

```
GET /section/page.html↵
```

Синтаксис носит фиксированный характер, за исключением, конечно, запрашиваемого ресурса (`/section/page.html`). В этой версии еще не существовало концепции полей заголовка (так называемых HTTP-заголовков), а также она не могла работать с медиафайлами, такими как, например, изображения. Сложно представить, что из такого простого протокола формата запрос–ответ, созданного для оптимизации поиска информации в одном исследовательском институте, вскоре возникла та самая Всемирная сеть, со всем ее многообразием информации, без которой уже невозможно представить современный мир. Еще на ранней стадии разработки своего изобретения Бернерс-Ли дал ему название Всемирная паутина (однако она очень отличалась от той, к которой мы привыкли), еще раз продемонстрировав свое предвидение масштабов проекта и планов превращения его в глобальную систему.

1.3.2 HTTP/1.0

Всемирная паутина возымела почти мгновенный успех. По данным NetCraft[1], к сентябрю 1995 года в ней насчитывалось уже 19 705 хостов. Спустя месяц их количество возросло до 31 568 и с тех пор продолжало увеличиваться бешенными темпами. На момент написания нашей книги количество существующих веб-сайтов приближается к 2 млрд. К 1995 году стало ясно, что функционала простейшего HTTP/0.9 уже недостаточно, а большинство веб-серверов реализовало расширения, выходящие

---

[1] https://news.netcraft.com/archives/category/web-server-survey/.

далеко за рамки его спецификации. Рабочая группа The HTTP Working Group (HTTP WG), возглавляемая Дейвом Рраггеттом (Dave Raggett), начала работать над HTTP/1.0 в попытке задокументировать «общее использование протокола». В мае 1996 года рабочей группой IETF был опубликован документ под названием RFC 1945[1] (Request for Comments – «Рабочее предложение»). Некоторые считают его официальным стандартом, однако многие вовсе так не думают[2]. Кроме того, существует еще одна версия RFC уже для HTTP/1.0, которая не является официальной спецификацией. Данный документ представлен нам как «памятка для Internet-сообщества, не задающая какой-либо стандарт».

Независимо от спорного состояния своего статуса RFC для HTTP/1.0 добавил в протокол некоторые ключевые обновления, такие как:

- дополнительные запросы `HEAD` и `POST`, помимо уже существующего `GET`;
- указание номеров HTTP-версии. Изначально предполагалось, что по умолчанию будет использоваться HTTP/0.9 для реализации обратной совместимости;
- HTTP-заголовки, которые использовались как в запросах, так и в ответах и должны были предоставлять больше информации о запрашиваемом ресурсе и отправляемом ответе;
- трехзначный код ответа, который указывал, был ли ответ успешным. Подобные коды свидетельствовали о запросах перенаправления, условных запросах и отображали статус ошибки (например, один из самых известных кодов `404 - Not Found`).

Такие усовершенствования были очень важны, поскольку возникли из реальной необходимости. HTTP/1.0 был создан не столько для внедрения каких-либо новых опций, сколько для того, чтобы задокументировать изменения, уже произошедшие в работе веб-серверов. Такие изменения открыли для Internet множество новых возможностей. Например, пользователи получили возможность добавлять на веб-страницы медиафайлы посредством заголовков HTTP-ответов, позволяющих определить тип содержимого данных в теле страницы.

Методы HTTP/1.0

Метод `GET` остался почти таким же, как и в HTTP/0.9. Однако в HTTP/1.0 появились заголовки, что позволило создавать `GET`-запросы с условием, с помощью которых клиент может уточнить, что хочет получить ресурс, только если он изменился с момента последнего получения – если страница не была изменена, клиент получает соответствующий ответ и продолжает использовать ранее полученную копию ресурса.

С появлением метода `HEAD` клиенты смогли получать все метаданные ресурса (например, HTTP-заголовки), не загружая при этом сам ресурс.

---

[1] https://tools.ietf.org/html/rfc1945.

[2] Замечательную публикацию про RFC можно найти по адресу https://www.mnot.net/blog/2018/07/31/read_rfc.

Этот метод полезен по многим причинам. Например, поисковая система, такая как Google, может проверить, был ли ресурс обновлен, и загрузить его уже в обновленном виде, что сэкономит ресурсы для обеих сторон.

Появление метода `POST` позволило клиенту отправлять данные непосредственно на веб-сервер. Таким образом, пользователи могли работать с файлом непосредственно через HTTP (при условии, что веб-сервер настроен на получение данных), вместо того чтобы выгружать новый HTML-файл на сервер, используя стандартные методы передачи данных. Метод `POST` может использоваться как при работе с целыми файлами, так и с небольшими фрагментами произвольных данных. Обычно этот метод используется в веб-формах для ввода данных на различных веб-сайтах, где содержимое веб-формы отправляется как пара «поле–значение» в виде HTTP-запроса. Таким образом, метод `POST` позволяет клиенту отправлять информацию на сервер в виде HTTP-запроса, имеющего собственное тело, как у HTTP-ответов.

К слову, с помощью `GET` вы можете отправлять данные посредством параметров запроса прямо в URL-адресе, поместив их после знака «?». Например, запрос в виде https://www.google.com/?q=search+string сообщит поисковой системе Google, что вас интересует `search string`. Параметры запроса были включены уже в самую раннюю спецификацию[1] унифицированного указателя ресурса (URI, Uniform Resource Locator), но они предназначались для уточнения URI путем предоставления дополнительных параметров, а в качестве способа загрузки данных на веб-сервер их не использовали. URL-адреса ограничены в длине и содержании (например, в них не могут быть использованы двоичные данные). URL-адрес не должен содержать конфиденциальные данные (пароли, данные кредитных карт и т. д.), так как в таком случае их можно будет увидеть на экране и в истории браузера. Таким образом, метод `POST` является более безопасным способом отправки данных, так как он обеспечивает конфиденциальность личных данных (тем не менее следует быть осторожным при отправке таких данных по обычному HTTP-соединению, а не по защищенному HTTPS, о чем мы поговорим позже). Еще одно отличие заключается в том, что `GET`-запрос *идемпотентен*, а `POST`-запрос таковым не является. Это означает, что при отправке нескольких `GET`-запросов на один и тот же URL-адрес, ответы на все запросы будут одинаковыми, а при отправке нескольких `POST`-запросов на один и тот же URL-адрес так случается не всегда. Например, когда вы обновляете веб-страницу, после обновления она должна остаться в неизменном виде. А когда вы обновляете страницу с подтверждением оплаты на торговой онлайн-площадке, браузер уточнит: «Вы уверены, что хотите отправить данные повторно? Это может привести к совершению дополнительной покупки» (хотя подобные веб-сайты должны гарантировать, что такого не случится!).

---

[1] https://tools.ietf.org/html/rfc1630.

Заголовки HTTP-запросов

В версии HTTP/0.9 для осуществления `GET`-запроса была отведена единственная строка, а в версии HTTP/1.0 появились заголовки. Они позволили через запрос предоставлять серверу дополнительную информацию, которая помогла бы обработать запрос эффективнее. Для HTTP-заголовков предназначены отдельные строки после начальной строки запроса. Таким образом, HTTP-запрос `GET` будет выглядеть не как

```
GET /page.html↵
```

а как

```
GET /page.html HTTP/1.0↵
Header1: Value1↵
Header2: Value2↵
↵
```

или без заголовков:

```
GET /page.html HTTP/1.0↵
↵
```

То есть к начальной строке был добавлен опциональный раздел *версии* (по умолчанию HTTP/0.9), а за опциональным разделом заголовка HTTP следовали два символа возврата каретки (новой строки, далее для краткости называемые *символами возврата*) в конце вместо одного. Второй символ возврата был необходим для отправки пустой строки, которая указывала на завершение опционального раздела заголовка запроса.

Заголовки HTTP имеют следующий вид: имя заголовка, двоеточие и далее содержимое заголовка. Согласно спецификации имя заголовка (не содержимое) не чувствительно к регистру. Заголовки можно разместить на нескольких строках, но тогда каждую новую строку нужно начинать с пробела или табуляции. Но так делать не рекомендуется, потому что не все клиенты или серверы используют этот формат и поэтому могут обработать заголовки неправильно. Вместо этого можно отправить несколько заголовков одного типа, которые семантически будут идентичны отправке версий, разделенных запятыми. Таким образом,

```
GET/page.html HTTP/1.0↵
Header1: Value1↵
Header1: Value2↵
```

обрабатывается также, как и

```
GET /page.html HTTP/1.0↵
Header1: Value1, Value2↵
```

В спецификации HTTP/1.0 прописано несколько стандартных заголовков, но на вышеприведенном примере мы можем видеть, что использование настраиваемых заголовков (в данном примере `Header1`), не зависит от того, какую версию протокола вы используете. Протокол был разработан так, чтобы его можно было обновлять и улучшать. Однако

в спецификации прямо говорится, что «эти поля не могут считаться распознаваемыми получателем» и могут быть проигнорированы, в то время как стандартные заголовки должны обрабатываться сервером, совместимым с HTTP/1.0.

Типичный `GET`-запрос HTTP/1.0 выглядит так:

```
GET /page.html HTTP/1.0↵
Accept: text/html,application/xhtml+xml,image/jxr/,*/*↵
Accept-Encoding: gzip, deflate, br↵
Accept-Language: en-GB,en-US;q=0.8,en;q=0.6↵
Connection: keep-alive↵
Host: www.example.com↵
User-Agent: MyAwesomeWebBrowser 1.1↵
↵
```

В данном примере мы можем видеть, что серверу сообщается информация о форматах, в которых вы можете принять ответ (HTML, XHTML, XML и т. д.), о том, что вы можете принять различные кодировки (например, алгоритмы сжатия данных, передаваемых по HTTP, такие как gzip, deflate и Brotli), а также о том, какие языки вы предпочитаете (британский английский, далее по приоритету американский английский и остальные формы английского языка) и о том, какой браузер вы используете. Также серверу сообщается о том, что соединение необходимо сохранять открытым (об этом мы поговорим позже). В конце запроса ставится два символа возврата. С этого момента мы не будем использовать эти символы в тексте нашей книги, чтобы он был более читабельным. Но имейте в виду, что в завершении последней строки запроса они обязательно должны быть.

Коды HTTP-ответов

Типичный ответ от сервера, использующего HTTP/1.0, выглядит следующим образом:

```
HTTP/1.0 200 OK
Date: Sun, 25 Jun 2017 13:30:24 GMT
Content-Type: text/html
Server: Apache

<!doctype html>
<html>
<head>
…и т. д.
```

Остальная часть кода HTML представлена следующим образом. Как видите, первая строка ответа включает информацию о HTTP-версии ответного сообщения (`HTTP/1.0`), трехзначный код состояния HTTP (`200`) и текстовое описание кода состояния (`OK`). Коды состояния и описания появились в HTTP/1.0. В HTTP/0.9 такого понятия, как код ответа, не существовало, а ошибки могли встретиться только в возвращаемом HTML-документе. В табл. 1.1 приведены коды HTTP-ответов согласно спецификации HTTP/1.0.

Таблица 1.1 Коды HTTP-ответов согласно спецификации HTTP/1.0

Категория	Код	Наименование	Описание
1xx (информационные)	нет	нет	HTTP/1.0 не определяет никаких кодов состояния 1xx, но определяет категорию
2xx (успешно)	200	OK	Стандартный код ответа для успешного запроса
	201	Created	Код должен быть возвращен по POST-запросу
	202	Accepted	Обработка запроса в процессе
	204	No content	Запрос успешно принят и обработан, но в ответе нет тела сообщения
3xx (перенаправление)	300	Multiple choices	Данный код используется редко. В нем говорится, что категория 3xx подразумевает, что ресурс доступен в одном (или нескольких) местах, а точный ответ предоставляет более подробную информацию о том, где он находится
	301	Moved permanently	Заголовок `Location` HTTP-ответа должен предоставить новый URL ресурса
	302	Moved temporarily	Заголовок `Location` HTTP-ответа должен предоставить новый URL ресурса
	304	Not modified	Используется для условных ответов, в которых тело не нужно отправлять снова
4xx (ошибка клиента)	400	Bad request	Запрос не может быть обработан, исправьте ошибку и запросите заново
	401	Unauthorized	Этот код показывает, что вы не авторизированы
	403	Forbidden	Вы авторизированы, но ваши идентификационные данные не имеют прав доступа
	404	Not found	Возможно, самый узнаваемый код HTTP-статуса, так как часто встречается на странице ошибки
5xx (ошибка сервера)	500	Internal server error	Запрос не удалось выполнить из-за ошибки сервера
	501	Not implemented	Сервер не распознает запрос (чаще всего из-за метода HTTP, который неизвестен серверу)
	502	Bad gateway	Сервер, выступая в роли шлюза или прокси-сервера, получил сообщение об ошибке от вышестоящего сервера
	503	Service unavailable	По техническим причинам, например перегрузка, сервер временно не может обрабатывать запросы

Внимательные читатели могут заметить, что некоторые коды (`203`, `303`, `402`) из более ранних версий HTTP / 1.0 RFC здесь отсутствуют. Некоторые дополнительные коды были исключены из окончательного опубликованного RFC. Однако несколько из них вернулось в спецификации в HTTP/1.1, но уже с другими описаниями и значениями. Администрация адресного пространства Internet (Internet Assigned Numbers Authority, IANA) поддерживает полный список кодов состояния HTTP во всех версиях HTTP. Коды состояния, представленные в табл. 1.1, впервые были определены в HTTP/1.0[1] и сейчас используются наиболее часто.

В некоторых случаях ответы могут совпадать. Например, какой код ответа вы получите в случае, если запрос не будет распознан сервером, `400` (неверный запрос) или `501` (запрос не реализован)? Существует большое разнообразие категорий кодов ответа, так что каждое приложение мо-

1 https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml.

жет использовать наиболее подходящую из них. В спецификации сказано, что список кодов ответов можно пополнить, поэтому по мере необходимости можно добавлять новые коды и для этого не обязательно менять сам протокол. К слову, это еще одна причина, по которой коды ответов делятся на категории. Новый код ответа (например, `504`) может быть не распознан существующим HTTP/1.0 клиентом, однако ему будет понятно, что по какой-то причине на стороне сервера произошла ошибка, и сможет обработать его так же, как он обрабатывает другие коды ответа категории 5xx.

Заголовки HTTP-ответов

После первой строки возврата идет определенное количество (ноль или более) строк ответа заголовка HTTP/1.0. Заголовки запросов и ответов формируются согласно одному и тому же формату. За ними следуют два символа возврата, а затем содержимое тела (жирным шрифтом):

```
GET /
HTTP/1.0 302 Found
Location: http://www.google.ie/?gws_rd=cr&dcr=0&ei=BWe1WYrf123456qpIbwDg Cache-
Control: private
Content-Type: text/html; charset=UTF-8
Date: Sun, 10 Sep 2017 16:23:33 GMT
Server: gws
Content-Length: 268
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
    <H1>302 Moved</H1>
        The document has moved
<A HREF="http://www.google.ie/?gws_rd=cr&amp;dcr=0&amp;ei=BWe1WYrfIojUgAbqpIbwDg">h
ere</A>.</BODY></HTML> Connection closed by foreign host.
```

Если предыдущая опубликованная версия HTTP/0.9 позволяла лишь извлекать статические документы из репозитория, то с появлением новой версии HTTP/1.0 синтаксис HTTP значительно расширился, и теперь стало возможным создание динамических, многофункциональных приложений. Кроме того, HTTP стал сложнее, а объем спецификации протокола расширился от 700 (как было в HTTP/0.9) до 20 000 слов (в HTTP/1.0 RFC). Однако даже после публикации новой спецификации рабочая группа HTTP Working Group рассматривала ее как временную версию, основной функцией которой было задокументировать текущее использование протокола, и уже работала над версией HTTP/1.1. Как уже было сказано, HTTP/1.0 был опубликован в целях создания стандартов и документации для HTTP, так как до этого протокол не был оформлен должным образом, а его синтаксис, которым могли бы пользоваться клиенты и серверы, не был определен. К стандарту RFC прилагались новые коды ответов, другие методы, такие как `PUT`, `DELETE`, `LINK`, и `UNLINK`, и дополнительные HTTP-заголовки, некоторые из которых позже войдут в стан-

дарт HTTP/1.1. HTTP Working Group с трудом справлялась с реализацией протокола и обнародовала ее только спустя пять лет.

1.3.3 HTTP/1.1

Как мы уже знаем, версия HTTP/0.9 предназначалась в основном для получения текстовых документов. Затем версия была расширена до HTTP/1.0, и ее функциональность вышла далеко за пределы текстового формата. Позже эта версия была дополнительно стандартизирована и уточнена в HTTP/1.1. Согласно системе управления версиями, HTTP/1.1 была по большей части модификацией HTTP/1.0 и не несла в себе радикальных изменений структуры протокола. Переход от 0.9 к 1.0 включал в себя гораздо больше изменений, так как были созданы HTTP-заголовки. HTTP/1.1 внес некоторые дополнительные улучшения и позволил оптимизировать использование протокола HTTP (например, постоянные соединения, обязательные заголовки сервера, улучшенные параметры кеширования и фрагментированное кодирование). Но – что, наверное, важнее всего – на ее основе был создан официальный стандарт, на котором строилось будущее Всемирной паутины. Несмотря на то что HTTP достаточно прост для понимания, существует множество тонкостей, которые могут быть реализованы несколько по-разному, а отсутствие формального стандарта затрудняет работу с ними.

Первая спецификация HTTP/1.1 была опубликована в январе 1997[1] года (всего через девять месяцев после публикации спецификации HTTP/1.0). Спецификации обновлялись два раза: в июне 1999-го[2] и в июне 2014-го[3]. После каждого обновления предыдущие версии считались устаревшими. Спецификация HTTP /1.1 составляла 305 страниц и содержала почти 100 000 слов. Исходя из этого можно увидеть, насколько расширился этот простой протокол и насколько важно было прояснить тонкости использования HTTP. Фактически на момент написания этой книги спецификация снова обновляется[4], и ожидается, что это обновление будет опубликовано в начале 2019 года (к моменту подготовки перевода обещанное большое обновление так и не появилось на сайте w3.org. – *Прим. ред.*). По своей сути, HTTP/1.1 не слишком отличается от HTTP/1.0. В первые два десятилетия своего существования Internet стремительно рос и расширялся, ввиду чего и стало необходимым создание документации, которая включала бы в себя информацию о принципах работы с новыми возможностями.

Описанию всех аспектов HTTP/1.1 может быть посвящена целая книга, а здесь мы попытаемся рассмотреть основные моменты – сформировать фон и контекст для дальнейшего рассмотрения HTTP/2. При переходе от HTTP/1.0 к HTTP/1.1 фундаментальная структура протокола не измени-

---

1 https://tools.ietf.org/html/rfc2068.

2 https://tools.ietf.org/html/rfc2616.

3 https://tools.ietf.org/html/rfc7230 и https://tools.ietf.org/html/rfc7235.

4 https://github.com/httpwg/http-core.

лась, а многие дополнительные функции HTTP/1.1 были созданы с помощью добавления HTTP-заголовков в HTTP/1.0. Однако в синтаксисе протокола произошли и некоторые серьезные изменения, например наличие заголовка хоста стало обязательным, а также были введены постоянные соединения.

Обязательный заголовок HOST

URL-адрес в строках HTTP-запроса (например, команда `GET`), является не абсолютным URL-адресом (например, `http://www.example.com/section/page.html`), а относительным URL (например, `/section/page.html`). При создании HTTP предполагалось, что веб-сервер будет содержать только один веб-сайт, хотя, возможно, на этом сайте будет много разделов и страниц. Прежде чем совершать HTTP-запросы, пользователь должен подключиться к веб-серверу, таким образом задается хост-часть URL-адреса. В настоящее время на многих веб-серверах может быть расположено сразу несколько сайтов (*виртуальный хостинг*), поэтому важно сообщить серверу, *какой именно сайт* вы хотите открыть, а также какой *относительный URL-адрес* вам нужен. Эту функцию можно было бы реализовать, изменив URL-адрес в HTTP-запросах на полный, абсолютный URL-адрес, но считалось, что многие существующие веб-серверы и клиенты не смогут распознать их. Вместо этого проблема была решена путем добавления заголовка `Host`:

```
GET / HTTP/1.1
Host: www.google.com
```

В отличие от HTTP/1.0 в HTTP/1.1 его наличие стало обязательным. Следующий запрос сформирован технически некорректно, так как он указывает на версию (HTTP/1.1), но не содержит заголовок `Host`:

```
GET / HTTP/1.1
```

Согласно спецификации[1] HTTP/1.1 этот запрос должен быть отклонен сервером (с кодом ответа 400), хотя сейчас большинство веб-серверов имеет хост для таких запросов по умолчанию. Введение обязательного заголовка `Host` стало важным шагом в HTTP/1.1. Он позволил серверам более широко использовать виртуальный хостинг и тем самым позволил Internet развиваться, не прибегая к созданию отдельных веб-серверов для каждого сайта. Кроме того, относительно низкие ограничения IP-адресов версии IPv4 были бы достигнуты гораздо раньше, если бы это изменение не вошло в силу. С другой стороны, если бы эти ограничения не было реализованы, возможно, они помогло бы форсировать переход на IPv6, который на момент написания этой книги существует уже более 20 лет и все еще находится в процессе развертывания.

Тот факт, что вместо изменения относительного URL-адреса на абсолютный в спецификации прописывалось наличие обязательного поля

1 https://tools.ietf.org/html/rfc7230#section-5.4.

заголовка Host, привел к появлению разногласий[1]. HTTP-прокси, введенные вместе с HTTP/1.1, позволяли подключаться к HTTP-серверу через посреднический HTTP-сервер. Синтаксис прокси-серверов требовал полных абсолютных URL-адресов для всех запросов, но на действующих веб-серверах (так называемых *серверах-источниках*) использовались заголовки Host. Как мы уже знаем, такой принцип работы помогал избежать прекращения работы существующих серверов, однако после того, как он стал обязательным, стало очевидно что после перехода на HTTP/1.1 и в целях лучшей совместимости с его реализациями все клиенты и серверы без исключения должны использовать запросы в стиле виртуального хостинга. В спецификации HTTP/1.1 сказано: «Для дальнейшего перехода к абсолютной форме URL-адресов для всех запросов в последующих версиях HTTP веб-серверы должны принимать абсолютный URL в запросах, даже если клиенты HTTP/1.1 будут отправлять их только прокси-серверам». Тем не менее, как мы увидим позже, в версии HTTP/2 эта проблема не была полностью решена. В ней вместо заголовка Host ввели поле для псевдозаголовка authority (см. главу 4).

Постоянные соединения (Keep-Alive)

В HTTP/1.0 были введены постоянные соединения. Это изменение было довольно значимым и, кроме того, поддерживалось многими серверами HTTP/1.0, несмотря на то что в спецификации этой версии постоянных соединений еще не было. Изначально HTTP представлял собой примитивный протокол типа «запрос–ответ». Клиент устанавливает соединение, запрашивает ресурс, получает ответ, и соединение закрывается. Сеть росла, и в ней появлялось все больше информации различных типов, в связи с чем закрытие соединений значительно снижало эффективность работы. Отображение одной страницы требовало использования нескольких HTTP-ресурсов, поэтому закрытие соединения (которое потом снова нужно будет открыть) вызывало ненужные задержки. Проблема была решена путем введения нового HTTP-заголовка Connection, совместимого с версией HTTP/1.0. Значение Keep-Alive в этом заголовке позволяет запросить сервер сохранить соединение открытым, чтобы разрешить отправку дополнительных запросов:

```
GET /page.html HTTP/1.0
Connection: Keep-Alive
```

Если сервер поддерживает постоянные соединения, он будет отвечать как обычно, но уже с использованием заголовка Connection: Keep-Alive в ответах:

```
HTTP/1.0 200 OK
Date: Sun, 25 Jun 2017 13:30:24 GMT
Connection: Keep-Alive
Content-Type: text/html
```

[1] С дискуссией по этому поводу можно ознакомиться по адресу https://lists.w3.org/Archives/Public/ietf-http-wg-old/1999SepDec/0014.html.

```
Content-Length: 12345
Server: Apache

<!doctype html>
<html>
<head>
…и т. д.
```

Получив такой ответ, клиент увидит, что он может отправить другой запрос на то же соединение после получения текущего ответа, поэтому серверу не нужно закрывать соединение с клиентом только для того, чтобы потом снова открыть его. При использовании постоянных соединений понять, когда ответ завершен, бывает довольно сложно; закрытие соединения является отчетливым признаком того, что сервер мог отправить ответ несуществующему соединению! HTTP-заголовок `Content-Length` предназначен для определения длины тела ответа. Когда ответ получен полностью, клиент может приступать к отправке следующего запроса.

HTTP-соединение может быть закрыто в любой момент как клиентом, так и сервером. Закрытие может произойти случайно (из-за сбоев сетевого подключения) или намеренно (например, если соединение некоторое время не используется, сервер закроет его, чтобы освободить ресурсы для других соединений). Поэтому даже при использовании постоянных соединений и клиенты, и серверы должны отслеживать их состояние и иметь возможность возобновить их в случае неожиданного закрытия. С некоторыми запросами ситуация усложняется. Например, если вы регистрируетесь на веб-сайте торговой площадки, не следует отправлять повторный запрос, если вы не удостоверились, что сервер обработал предыдущий.

В версии HTTP/1.1 процесс постоянного подключения был добавлен в официальный стандарт и с тех пор выполняется по умолчанию. Любое HTTP/1.1 соединение носит постоянный характер, даже если ответы не содержат заголовка `Connection: Keep-Alive`. Если сервер намерен закрыть соединение, он должен включить в ответ заголовок `Connection: close`:

```
HTTP/1.1 200 OK
Date: Sun, 25 Jun 2017 13:30:24 GMT
Connection: close
Content-Type: text/html; charset=UTF-8
Server: Apache

<!doctype html>
<html>
<head>
…и т. д.
Connection closed by foreign host.
```

Ранее в этой главе мы рассматривали эту тему на примерах работы с Telnet. Теперь попробуйте снова отправить через Telnet следующее:

- запрос HTTP/1.0 без заголовка `Connection: Keep-Alive`. Соединение будет автоматически закрыто сервером после отправки ответа;

- тот же запрос HTTP/1.0, но с заголовком `Connection: Keep-Alive`. Соединение остается открытым;
- запрос HTTP.1.1 с заголовком `Connection: Keep-Alive` или без него. Соединение остается открытым по умолчанию.

Нет ничего необычного в том, что клиенты HTTP/1.1 включают заголовок `Connection: Keep-Alive` в запросы HTTP/1.1 в явном виде, несмотря на то что он используется по умолчанию. Точно так же серверы иногда включают заголовок в HTTP/1.1 ответы, несмотря на то что это не обязательно.

Таким образом, если веб-браузер обрабатывает HTML-документ и видит, что ему нужен файл CSS и файл JavaScript, он должен иметь возможность отправлять запросы на эти файлы одновременно и получать ответы обратно в нужном порядке, а не ждать первого ответа перед отправкой второго запроса. Вот пример:

```
GET /style.css HTTP/1.1
Host: www.example.com

GET /script.js HTTP/1.1
Host: www.example.com

HTTP/1.1 200 OK
Date: Sun, 25 Jun 2017 13:30:24 GMT
Content-Type: text/css; charset=UTF-8
Content-Length: 1234
Server: Apache

.style {
…и т. д.

HTTP/1.1 200 OK
Date: Sun, 25 Jun 2017 13:30:25 GMT
Content-Type: application/x-javascript; charset=UTF-8
Content-Length: 5678
Server: Apache

Function(
…и т. д.
```

К сожалению, существует ряд причин, по которым технология конвейеризации запросов не развивалась, а клиенты и серверы довольно плохо поддерживают ее. Об этом мы поговорим во второй главе. Короче говоря, возможность повторно использовать постоянное TCP-соединение для нескольких запросов сильно увеличивает производительность, однако в большинстве реализаций HTTP/1.1 она по-прежнему не используется. Во время обработки одного запроса для других запросов HTTP-соединение недоступно.

Другие новые функции

В спецификацию HTTP/1.1 было введено множество новых функций:

- к методам `GET`, `POST` и `HEAD` (введенных еще в HTTP/1.0) добавились новые. Среди них `PUT`, `OPTIONS` и менее используемые `CONNECT`, `TRACE`, и `DELETE`;

- методы кеширования. Они позволяли серверу поручить клиенту сохранить ресурс (например, CSS-файл) в кеше браузера, чтобы он мог быть повторно использован позже, если потребуется. HTTP-заголовок `Cache-Control`, представленный в HTTP/1.1, имел больше параметров, чем заголовок `Expires` из HTTP/1.0;
- файлы HTTP-куки, благодаря которым HTTP перестал быть протоколом без сохранения состояния;
- объявление рабочей кодировки (как показано в некоторых примерах в этой главе) и языка HTTP-ответов;
- поддержка прокси;
- аутентификация;
- новые коды состояния;
- завершающие заголовки (которые мы обсудим в главе 4, раздел 4.3.3).

В целях расширения возможностей к HTTP постоянно добавляются новые заголовки, многие из которых обеспечивают лучшую производительность или безопасность. Спецификация HTTP/1.1 не претендует на звание окончательной и поощряет появление новых заголовков. В ней даже есть раздел[1], посвященный тому, как заголовки должны быть определены и задокументированы. Как уже упоминалось, некоторые из этих заголовков добавляются по соображениям безопасности и используются для того, чтобы веб-сайт мог сообщить браузеру о включении определенных дополнительных средств защиты, поэтому они не требуют реализации на стороне сервера (кроме возможности отправки заголовка). Одно время существовало соглашение, согласно которому к нестандартизированным заголовкам должен был добавляться префикс `X-` (`X-Content-Type`, `X-Frame-Options`, `X-XSS-Protection`). Сейчас это соглашение устарело[2], и новые экспериментальные заголовки трудно отличить от заголовков, прописанных в спецификации HTTP/1.1. Нередко такие заголовки закрепляются в локальных RFC (`Content-Security-Policy`[3], `Strict-Transport-Security`[4], и т. д.).

1.4 Введение в HTTPS

Изначально, HTTP был простым текстовым протоколом. HTTP-сообщения передаются через Internet в незашифрованном виде и поэтому они могут быть перехвачены по пути к месту назначения. Как следует из самого названия, Internet представляет собой сеть, состоящую из множества компьютеров, каждый из которых может взаимодействовать друг с другом, а не систему с прямыми связями между двумя машинами. Фиксированной маршрутизации сообщений в Internet не существует.

1 https://tools.ietf.org/html/rfc7231#section-8.3.1.
2 https://tools.ietf.org/html/rfc6648.
3 https://tools.ietf.org/html/rfc7762.
4 https://tools.ietf.org/html/rfc6797.

После того как вы отправляете сообщение, оно совершает длинный путь от вашего Internet-провайдера (ISP) до телекоммуникационных сетей и принимающей стороны, поэтому вы не узнаете, сколько других компьютеров могут увидеть его. Поскольку HTTP-сообщение – это обычный текст, злоумышленникам не составит труда перехватить его, прочитать или даже изменить содержание.

HTTPS является защищенной версией HTTP, которая шифрует передаваемые сообщения с помощью протокола Transport Layer Security (TLS). Предыдущая версия носила название Secure Sockets Layer (SSL), и о ней вы можете прочесть в сноске ниже. HTTPS добавляет к HTTP три важных аспекта:

- *шифрование* (третьи лица не смогут прочитать сообщения во время их передачи);
- *целостность* (сообщение невозможно изменить в процессе передачи, так как зашифрованное сообщение имеет цифровую подпись, и эта подпись криптографически проверяется перед расшифровкой);
- *аутентификация* (обеспечивает передачу сообщения именно на нужный вам сервер).

SSL, TLS, HTTPS и HTTP

HTTPS зашифровывает сообщения по протоколам SSL и TLS. SSL был разработан компанией Netscape. Версия протокола SSLv1 так и не увидела свет, и вместо нее в 1995 году Netscape выпустила версию SSLv2. В 1996 году компания выпустила SSLv3, в которой были исправлены некоторые недостатки предыдущих версий.

SSL не считался официальным Internet-стандартом, поскольку принадлежал компании Netscape, но впоследствии IETF опубликовала его постфактум как официальный документ[a]. Далее на основании SSL был оформлен и стандартизирован новый протокол, получивший название TLS. Протоколы TLSv1.0[b] и SSLv3 были весьма похожи, однако все же несовместимы. Версии TLSv1.1[c] и TLSv1.2[d] были созданы в 2006 и 2008 годах соответственно. Они стали более безопасными. TLSv1.3 был признан стандартом в 2018 году[e]. Он более безопасен и эффективен[f], но потребуется еще много времени, чтобы он получил широкое распространение.

Несмотря на то что были созданы новые и более безопасные стандартизированные версии, многие люди все еще пользовались SSLv3, поэтому он долгое время оставался стандартом де-факто, хотя многие клиенты поддерживали и TLSv1.0. Однако в 2014 году в SSLv3 были обнаружены серьезные уязвимости[g], в связи с чем он перестал поддерживаться браузерами и вышел из употребления[h]. Эта ситуация положила начало развитию TLS. Спустя некоторое время в TLSv1.0 обнаружили аналогичные уязвимости. Совет по стандартам безопасности призывал использоватьTLSv1.1 или более поздние версии[i].

В итоге люди стали путать названия протоколов. Многие до сих пор говорят «SSL», потому что он оставался стандартом очень долгое время, а другие го-

ворят «SSL/TLS» или «TLS». Чтобы избежать недоразумений, иногда говорят просто «HTTPS», хотя этот термин и не совсем корректен. В данной книге мы называем шифрование «HTTPS» (а не SSL или SSL/TLS). Однако если речь идет конкретно о TLS, мы используем соответствующий термин. Следуя тому же принципу, когда мы говорим об основной семантике HTTP, мы говорим просто «HTTP» независимо от того, идет речь о незашифрованном или зашифрованном HTTPS-соединении.

---

[a] https://tools.ietf.org/html/rfc6101.

[b] https://tools.ietf.org/html/rfc2246.

[c] https://tools.ietf.org/html/rfc4346.

[d] https://tools.ietf.org/html/rfc5246.

[e] https://tools.ietf.org/html/rfc8446.

[f] https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/.

[g] https://www.us-cert.gov/ncas/alerts/TA14-290A.

[h] https://tools.ietf.org/html/rfc7568.

[i] https://www.pcisecuritystandards.org/documents/Migrating-from-SSL-Early-TLS-Info-Suppv1_1.pdf.

HTTPS работает с использованием шифрования с открытым ключом, которое позволяет серверам предоставлять пользователям открытые ключи в виде цифровых сертификатов при первом подключении. Ваш браузер шифрует сообщения с помощью этого открытого ключа, расшифровать который может только сервер-получатель, так как только он имеет соответствующий секретный ключ. Такая система обеспечивает вам безопасное взаимодействие с веб-сайтом без необходимости заранее знать общий секретный ключ. Такой подход особенно важен для таких систем, как Internet, где новые веб-сайты и пользователи появляются и исчезают ежедневно каждую секунду.

Цифровые сертификаты выдаются различными центрами сертификации (Certification authority), с которыми работают веб-браузеры, а также подписываются ими цифровой подписью, поэтому проверить подлинность открытого ключа для сервера, к которому вы подключаетесь, очень легко. Проблема заключается в том, что HTTPS показывает, что вы подключаетесь к серверу, но не гарантирует его безопасности. С помощью HTTPS для поддельных фишинговых сайтов можно легко установить другой, но похожий на оригинальный домен (exmplebank.com вместо examplebank.com). HTTPS-сайты обычно отмечены в веб-браузерах значком зеленого навесного замочка. Многие пользователи считают, что это означает, что *сайт безопасен*, однако на самом деле это просто говорит о том, что веб-сайт использует *безопасное шифрование*.

Некоторые центры сертификации проводят дополнительную проверку веб-сайтов при выдаче сертификатов, а также предоставляют расширенный сертификат проверки (известный как сертификат EV – Extended Validation), который обеспечивает шифрование HTTP-трафика так же, как и обычный сертификат, но в большинстве веб-браузеров он отображает название компании, как показано на рис. 1.4.

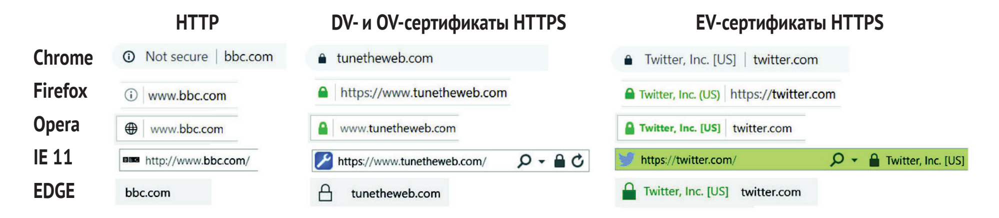


Рис. 1.4 HTTPS-индикаторы веб-браузера

Многие люди оспаривают преимущества сертификатов EV[1], главным образом потому что подавляющее большинство пользователей не замечает названия компании на сайтах и не видит разницы между сертификатами EV и стандартными сертификатами (Domain Validated, DV). Сертификаты проверенных организаций (Organization validation, OV) являются сертификатами среднего уровня безопасности, так как они подразумевают некоторые проверки при оформлении, но не дают дополнительных уведомлений в веб-браузерах, что делает их бессмысленными на техническом уровне (хотя центры сертификации могут предлагать дополнительные поддержку в рамках покупки таких сертификатов).

На момент написания данной книги команда Google Chrome все еще исследует такие индикаторы безопасности и экспериментирует с ними[2]. Они считают схему (`http` и `https`), префикс `www` и, возможно, даже сам значок замочка ненужными элементами и стараются скрыть их (кроме того, они считают, что использование HTTPS является нормой, а сайты, поддерживающие только HTTP, должны иметь пометку «небезопасный»). Команда также рассматривает вопрос о том, не стоит ли отказаться от сертификата EV[3].

Протокол HTTPS разработан на основе HTTP и очень схож с ним. По умолчанию у него другой порт (порт 443, в отличие от стандартного порта 80 для HTTP), а также он имеет другую схему URL-адресов (`https://`, а не `http://`). Однако эти аспекты не столь значительны, так как HTTPS и HTTP имеют одинаковый синтаксис и формат сообщений, а отличаются они лишь в планах шифрования и дешифрования.

Когда клиент подключается к HTTPS-серверу, он проходит через стадию согласования (или TLS-рукопожатия). Во время этого процесса сервер предоставляет открытый ключ, клиент и сервер согласовывают используемые методы шифрования, а затем клиент и сервер согласовывают общий ключ шифрования, чтобы использовать его в будущем. (Криптография с открытым ключом работает медленно, поэтому открытые ключи шифрования применяются только для согласования подключения сервера и клиента, который используется для шифрования

---

1 https://www.tunetheweb.com/blog/what-does-the-green-padlock-really-mean/.

2 https://blog.chromium.org/2018/05/evolving-chromes-security-indicators.html.

3 https://groups.google.com/forum/#!topic/mozilla.dev.security.policy/szD2KBHfwl8%5B1-25%5D.

будущих сообщений, что обеспечивает лучшую производительность.) Подробнее мы поговорим о TLS-рукопожатии в главе 4 (раздел 4.2.1).

После создания сессии HTTPS происходит обмен стандартными HTTP-сообщениями. Клиент и сервер зашифровывают их перед отправкой и расшифровывают при получении, однако для обычного веб-разработчика или менеджера сервера между HTTPS и HTTP нет уже фактически никакой разницы. Все процессы происходят прозрачно, если вы, конечно, не смотрите на необработанные сообщения, отправленные по сети. При переходе на HTTPS мы продолжаем пользоваться стандартными HTTP-запросами и ответами, а не заменяем их каким-то другим протоколом.

HTTPS – это обширная тема, которая выходит далеко за рамки этой книги. Однако мы снова затронем этот вопрос в следующих главах, поскольку с внедрением HTTP/2 ситуация несколько изменилась. Но сейчас нам необходимо знать только то, что HTTPS существует и что он работает на несколько ином уровне, нежели HTTP (между TCP и HTTP). Если вы не смотрите непосредственно на зашифрованные сообщения, вы не почувствуете никакой значительной разницы между HTTP и HTTPS.

Веб-серверам, использующим HTTPS, необходим клиент с поддержкой HTTPS, выполняющий шифрование и дешифрование. Telnet уже не подходит для этой задачи, поэтому для отправки примеров HTTP-запросов на эти серверы следует выбрать другой клиент. Вы можете использовать команду `s_client` в OpenSSL, которая позволит вам отправлять HTTP-команды на HTTPS-сервер аналогично тому, как вы делали это с помощью Telnet:

```
openssl s_client -crlf -connect www.google.com:443 -quiet
GET / HTTP/1.1
Host: www.google.com
HTTP/1.1 200 OK
…и т. д.
```

Между тем мы заканчиваем рассмотрение инструментов командной строки и переходим к изучению более эффективных способов работы с HTTP-запросами. В следующем разделе мы кратко рассмотрим инструменты веб-браузера, с помощью которых работать с HTTP-запросами и ответами становится намного проще.

1.5 Инструменты для просмотра, отправки и получения HTTP-сообщений

Безусловно, такие инструменты командной строки, как Telnet, отлично подходят для изучения основ HTTP, однако они имеют некоторые ограничения и не подходят для работы с веб-страницами больших размеров, каких, к слову, в Internet довольно много. Существуют инструменты, справляющиеся с этими задачами лучше, чем Telnet. Многими из них вы можете пользоваться непосредственно в вашем веб-браузере.

1.5.1 Использование инструментов разработчика в веб-браузерах

Сегодня все веб-браузеры оснащены так называемыми *инструментами разработчика*, с помощью которых вы можете увидеть некоторые детали устройства и работы веб-сайтов, включая HTTP-запросы и ответы.

Инструменты разработчика запускаются сочетанием клавиш (**F12** в большинстве браузеров для ОС Windows или **Option**+**Command**+**I** на компьютерах Apple). Так же вы можете кликнуть на элемент страницы правой кнопкой мыши и во всплывающем контекстном меню выбрать **Посмотреть код**. Панель инструментов разработчика содержит несколько вкладок, в которых вы можете увидеть различные технические детали строения веб-страниц, однако сейчас нас больше всего интересует вкладка **Network** (Сеть). Если вы откроете инструменты разработчика, а затем загрузите веб-страницу, во вкладке **Network** вы увидите все HTTP-запросы, а при нажатии на них получите более подробную информацию, в том числе заголовки запроса и ответа.

На рис. 1.5 вы можете увидеть панель инструментов разработчика в веб-браузере Chrome при загрузке https://www.google.com.

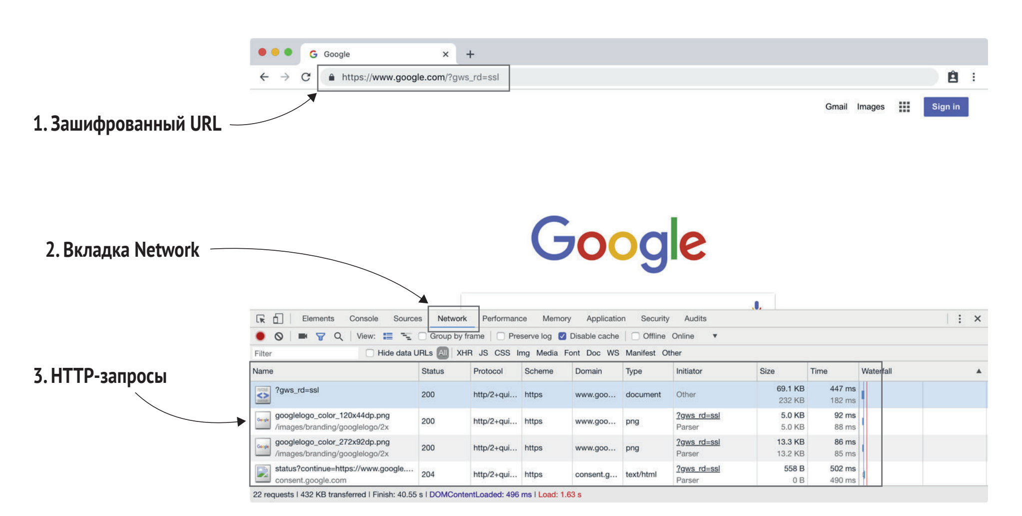


Рис. 1.5 Панель инструментов разработчика в Chrome

URL-адрес вводится в верхней части адресной строки (1). Обратите внимание на значок замочка и `https://`. Это означает, что Google использует HTTPS (хотя, как уже упоминалось, Chrome может перестать использовать его). Веб-страница размещается под адресной строкой. Однако если вы откроете панель инструментов разработчика, то увидите на странице новый раздел, содержащий различные вкладки. При нажатии на вкладку **Сеть** (2) отображаются HTTP-запросы (3), включая такую информацию, как HTTP метод (`GET`), статус ответа (`200`), версия протокола (`http/1.1`) и схема (`https`). Вы можете изменить отображаемые столбцы,

кликнув правой кнопкой мыши на их заголовки. Например, столбцы **Protocol** (Протокол), **Scheme** (Схема) и **Domain** (Домен) не отображаются по умолчанию, и на некоторых веб-сайтах (например, Twitter) в столбце для HTTP/2 вы можете увидеть `h2`, а для более новой версии протокола (которую мы рассмотрим в главе 9), возможно, даже `http/2+quic` (Google).

На рис. 1.6 показано, что происходит при нажатии на первый запрос (1). Справа вы можете увидеть панель вкладок с заголовками ответов (2) и заголовками запросов (3). Со многими из них (но не со всеми) мы уже познакомились в этой главе.

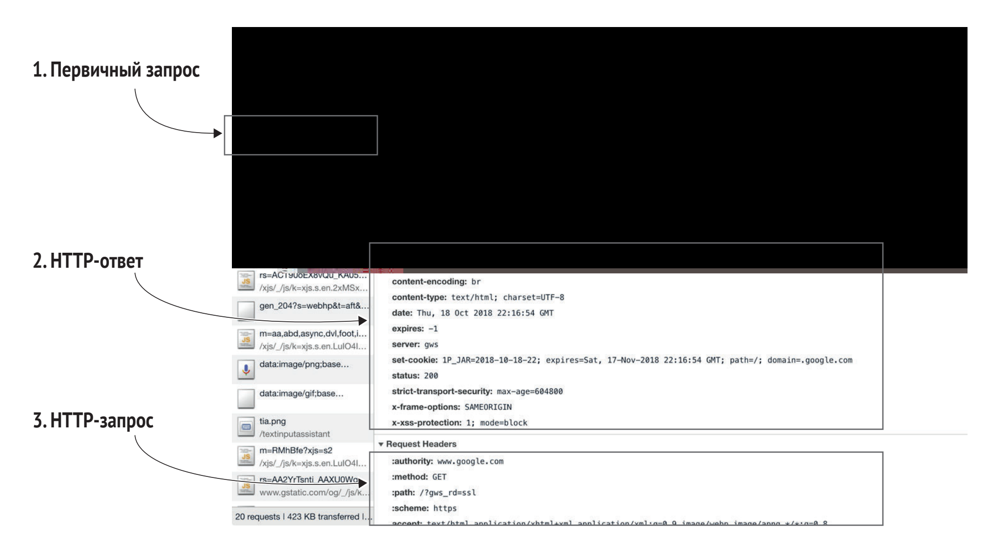


Рис. 1.6 Просмотр HTTP-заголовков в Chrome

Браузер обрабатывает HTTPS-запросы, поэтому инструменты разработчика показывают сообщения запросов до их шифрования и ответные сообщения после их дешифрования. Как правило, если у вас есть правильные инструменты для обработки шифрования и дешифрования сообщений, после настройки HTTPS-соединения дальнейшая работа протокола уже мало чем интересна. Кроме того, инструменты разработчика в большинстве веб-браузеров хорошо справляются с задачей отображения медиафайлов, а код (HTML, CSS, и JavaScript) можно отформатировать, чтобы его было легче читать. Периодически мы будем возвращаться к теме инструментов разработчика. Если вы еще не знакомы с инструментами разработчика вашего браузера для часто посещаемых вами сайтов, то настоятельно рекомендуем изучить их.

1.5.2 Отправка HTTP-запросов

Использование инструментов разработчика веб-браузеров является наилучшим способом просмотра еще не обработанных HTTP-запросов и ответов, однако – что весьма удивительно – отправлять такие запросы эти

инструменты не могут. Инструменты разработчика в браузере редко позволяют отправлять необработанные HTTP-сообщения, но, например, с помощью адресной строки можно отправлять простые `GET`-запросы, а некоторые веб-сайты предоставляют определенные функциональные возможности, например отправку `POST`-запросов с помощью HTML-форм.

Расширение для браузеров Advanced REST Client дает вам возможность отправлять необработанные HTTP-сообщения и просматривать ответы. Отправьте запрос `GET` (1) для URL-адреса https://www.google.com (2) и нажмите **Отправить** (3), затем вы получите ответ (4), как показано на рис. 1.7. Обратите внимание, что это приложение также обрабатывает запросы и ответы с помощью HTTPS.

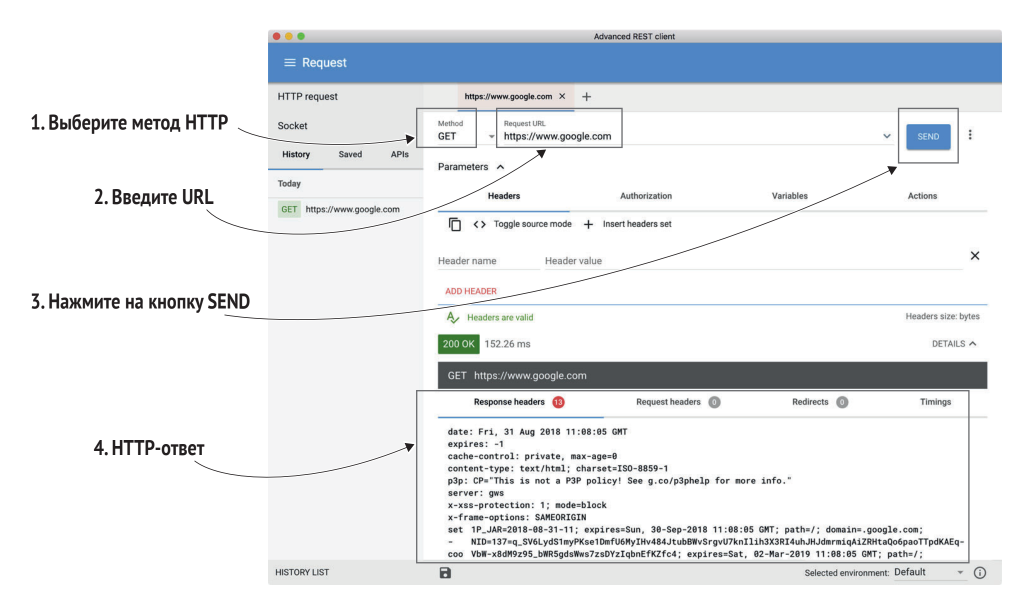


Рис. 1.7 Расширения для браузеров Advanced REST Client

Работа с расширением Advanced Rest Client очень проста и привычна, однако при этом оно позволяет отправлять многие типы HTTP-запросов (например, `POST` и `PUT`), а также создавать заголовок или тело данных, предназначенных для отправки. Изначально Advanced REST Client был расширением для Chrome, но позже его оформили как отдельное приложение. Существуют и другие подобные расширения, имеющие соответствующую функциональность, например Postman (Chrome), Rested, RESTClient (Firefox) и RESTMan (Opera).

1.5.3 Другие инструменты для просмотра и отправки HTTP-запросов

Вне браузера вам также доступно множество других инструментов для отправки или просмотра HTTP-запросов. Среди них и инструменты

командной строки (например, curl, wget и httpie), и настольные клиенты (например, SOAP-UI). Если вы хотите ознакомиться с сетевой информацией, зайдите на страницу net-internals в Chrome или воспользуйтесь анализаторами трафика, такими как Fiddler и Wireshark. О некоторых из таких инструментов мы поговорим в последующих главах, когда будем рассматривать детали HTTP/2, но на данный момент инструментов, упомянутых в этом разделе, должно быть достаточно.

Резюме

- HTTP является одной из основных технологий в Internet.
- Для того чтобы загрузить веб-страницу, веб-браузеры делают несколько HTTP-запросов.
- Изначально протокол HTTP был простым текстовым протоколом.
- За последние 20 лет HTTP стал сложнее, но все еще остался протоколом текстового формата.
- HTTPS шифрует стандартные HTTP-сообщения.
- Для просмотра и отправки HTTP-сообщений существуют различные инструменты.

2 Путь к HTTP/2

В этой главе мы рассмотрим:

- проблемы с производительностью, возникающие при использовании HTTP/1.1;
- пути решения таких проблем;
- практические примеры;
- протокол SPDY и его роль в оптимизации производительности HTTP/1;
- SPDY и стандартизацию HTTP/2;
- оптимизацию производительности в HTTP/2.

Зачем нам нужен HTTP/2? Ведь сайты, использующие HTTP/1, все равно прекрасно функционируют, не так ли? Что же такое HTTP/2 на самом деле? В этой главе мы ответим на все вопросы исходя из практических примеров, а также покажем, почему переход на HTTP/2 так необходим и почему многие владельцы сайтов еще не сделали этого.

На основе протокола HTTP/1.1 создана большая часть сайтов в Internet, и до определенного времени он достаточно хорошо выполнял свои функции (для технологии 20-летней давности). Однако сейчас количество Internet-пользователей значительно возросло, а сайты из простых статических страниц превратились в полностью интерактивные, благодаря чему мы можем совершать банковские операции, заниматься шопингом, бронировать путевки, посещать различные медиаплощадки, общаться и делать много других дел онлайн.

Internet становится доступнее и быстрее благодаря разработке новых технологий. Появление широкополосного доступа и подключение к сети Internet офисов и домов при помощи волоконно-оптических кабелей в разы увеличило скорость по сравнению с использованием устаревших коммутируемых соединений. Технологии 3G и 4G позволяют потребителям пользоваться высокоскоростным мобильным интернетом по разумным ценам.

Несмотря на значительное ускорение загрузки, потребность в увеличении скоростей возрастала еще быстрее. Вероятно, в течение некоторого времени скорость широкополосного доступа в Internet продолжит расти. Однако существуют некоторые ограничения, преодолеть которые весьма сложно. Как вы увидите позже, ключевым фактором скорости загрузки веб-страниц является задержка, которая напрямую зависит от скорости света в оптоволокне – универсальной постоянной, превысить которую не позволяют законы физики.

2.1 HTTP/1.1 и современная Всемирная паутина

Из главы 1 мы узнали, что HTTP является протоколом запроса и ответа, который изначально позволял совершать только одиночные запросы объектов с текстовым содержимым, после чего соединение закрывалось. С внедрением версии HTTP/1.0 появилась возможность наполнять веб-страницы различными типами мультимедиа, например, изображениями, а в версии HTTP/1.1 ввели режим «постоянного соединения», что позволило в случае необходимости совершать сразу несколько запросов за одно соединение.

Безусловно, тогда эти изменения были полезны, но с момента выхода последней версии HTTP (HTTP/1.1 в 1997 году; хотя, как было сказано в главе 1, официальная спецификация этой версии несколько раз уточнялась и во время написания нашей книги также находится в процессе уточнения) Internet значительно изменился. На сайте проекта HTTP Archive (https://httparchive.org/reports/state-of-the-web) вы можете проследить развитие сайтов за последние восемь лет, как показано на рис. 2.1. Не обращайте внимания на небольшой спад графика в мае 2017 года, который связан с проблем измерений в HTTP Archive[1].

Как вы можете заметить, в среднем веб-сайт совершает 80–90 запросов ресурсов и загружает около 1,8 Мб данных (учитывается объем данных, передаваемых по сети, включая текстовые ресурсы, сжатые с помощью gzip или аналогичных приложений). Размер страниц веб-сайтов, не использующих сжатие, составляет около 3 Мб, что вызывает проблемы при их загрузке на таких сетевых устройствах, как мобильные телефоны.

---

[1] https://github.com/HTTPArchive/legacy.httparchive.org/issues/98#issuecomment-301641938.

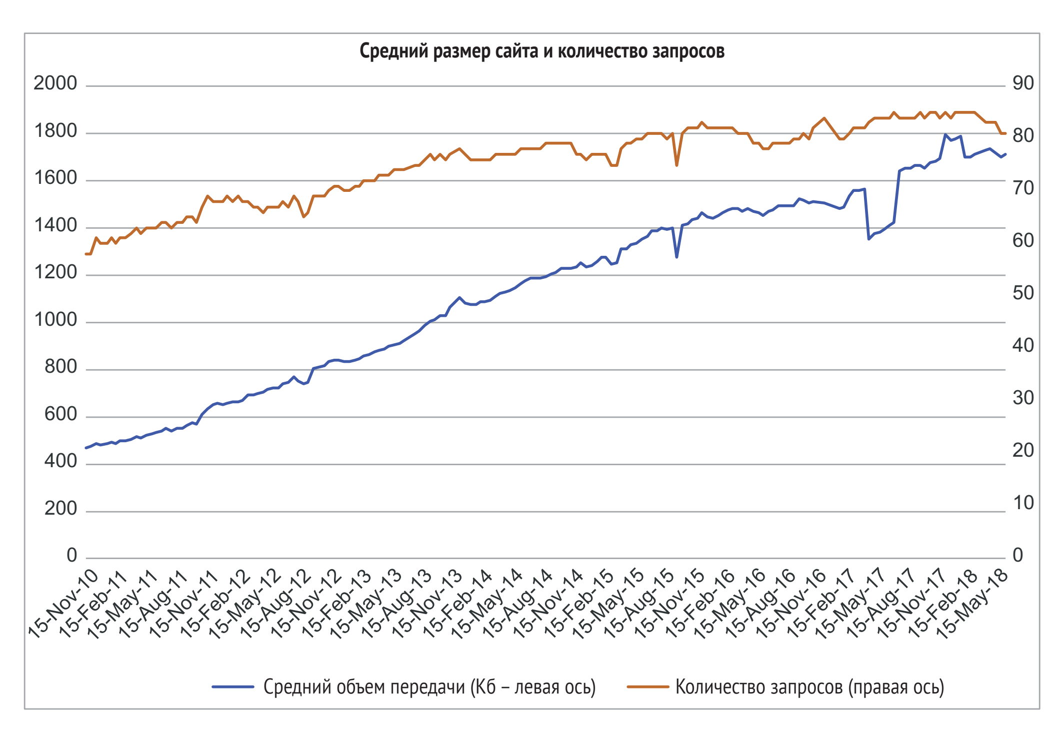


Рис. 2.1 Средний размер сайтов 2010–2018 гг.[1]

Однако этот средний показатель сильно варьируется. Например, в табл. 2.1 приведена статистика Alexa Top 10[2], в которую включены самые популярные сайты в Соединенных Штатах.

Таблица 2.1 Топ-10 веб-сайтов США по популярности

Популярность	Сайт	Количество запросов	Размер
1	https://www.google.com	17	0.4 MB
2	https://www.youtube.com	75	1.6 MB
3	https://www.facebook.com	172	2.2 MB
4	https://www.reddit.com	102	1.0 MB
5	https://www.amazon.com	136	4.46 MB
6	https://www.yahoo.com	240	3.8 MB
7	https://www.wikipedia.org	7	0.06 MB
8	https://www.twitter.com	117	4.2 MB
9	https://www.ebay.com	160	1.5 MB
10	https://www.netflix.com	44	1.1 MB

В таблице показано, что некоторые веб-сайты (такие как «Википедия» и Google) оптимизированы, поэтому не требуют большого количество ресурсов, однако многим другим сайтам необходимы сотни ресурсов и большое количество мегабайт данных. Таким образом, ценность этих

[1] https://httparchive.org/reports/state-of-the-web.

[2] https://www.alexa.com/topsites/countries/US.

средних статистических данных ставится под сомнение[1]. Тем не менее отчетливо наблюдается тенденция к увеличению объема данных и задействованию все большего количества ресурсов. Увеличение веса веб-сайтов обусловлено прежде всего тем, что они наполняются различными мультимедийными элементами. На большинстве веб-сайтов размещены какие-либо изображения и видео. Кроме того, сайты становятся более сложными, а для правильного отображения их совместимости используется множество фреймворков и зависимостей.

Изначально веб-страницы были статичны, но, по мере того как веб становился более интерактивным, на стороне серверов появилась возможность динамической генерации содержимого страниц посредством таких технологий, как *общий интерфейс запускаемых клиентом приложений* (Common Gateway Interface, CGI) и *Java Servlet/Java Server Pages* (JSP). На следующем этапе развития произошел переход от полного формирования страниц на стороне сервера к базовым страницам в формате HTML, созданным на основе технологии AJAX (*Asynchronous JavaScript and XML*), позволяющей осуществлять дополнительные запросы к серверу на стороне клиента. Такие запросы дают возможность изменения содержимого веб-страницы без необходимости полной перезагрузки страницы или динамической генерации копии содержимого на стороне сервера. Простейший способ понять, как это работает, – взглянуть на изменения в веб-поиске. На ранней стадии развития веба, еще до того, как появились первые поисковые системы, основными помощниками при поиске информации в Internet были статические веб-хранилища сайтов и страниц, которые к тому же обновлялись крайне редко. Затем возникли первые поисковые системы, с помощью которых пользователи отправляли поисковые формы и получали результаты от сервера (динамические страницы, генерируемые на стороне сервера). Сегодня на большинстве поисковых сайтов при вводе текста в поисковую строку еще до нажатия кнопки **Поиск** раскрывается меню, где пользователю предлагается несколько вариантов продолжения его запроса. В этом отношении компания Google пошла еще дальше и представила функцию показа запросов, введенных пользователями (летом 2017 года она была отменена, поскольку все больше поисковых запросов совершалось с мобильных устройств, и эта функция утратила свое значение).

Помимо поисковых систем, подход AJAX используют и другие виды веб-страниц, начиная от сайтов социальных сетей, на которые загружаются новые публикации, до новостных сайтов, где домашние страницы обновляются по мере поступления новостей. Благодаря наличию медиафайлов и использованию AJAX-запросов веб сайты превращаются в более интересные *веб-приложения*. Разработчики HTTP не могли предвидеть огромное увеличение количества ресурсов, а наличие проблем с производительностью объясняется простотой этого протокола.

---

1 https://speedcurve.com/blog/web-performance-page-bloat/.

2.1.1 Основные проблемы с производительностью HTTP/1.1

Представим, что у нас есть простая статическая веб-страница, на которой размещен текст и два изображения, а переход запроса к статическому веб-серверу через Internet занимает 50 миллисекунд (мс). В подобных случаях веб-сервер просто выбирает файл из файлового сервера и отправляет его обратно примерно за 10 мс. Также веб-браузеру требуется 10 мс для обработки изображения и отправки следующего запроса. Эти цифры гипотетичны – если вы пользуетесь *системой управления контентом* (Content mamagement system, CMS), позволяющей мгновенно создавать страницы (например, Wordpress для обработки страниц использует PHP), то 10 мс будет приблизительным значением, которое зависит от того, какой тип информации обрабатывается сервером и/или базой данных. Кроме того, изображения могут быть достаточно большими, и для их отправки потребуется больше времени, чем для отправки HTML-страницы. Позже мы рассмотрим реальные примеры различных ситуаций, а для вышеописанной ситуации пример передачи информации по HTTP-соединению будет выглядеть, как на рис. 2.2.

В рамках представлены стадии обработки на стороне клиента или сервера, а стрелки представляют собой сетевой трафик. На этом примере видно, сколько времени уходит на отправку сообщений туда и обратно. Из 360 мс, необходимых для отображения готовой страницы, только 60 мс было потрачено на обработку запросов со стороны клиента или браузера. 300 мс (более 80 % времени) было потрачено на ожидание передачи сообщений через Internet. В это время веб-браузер и веб-сервер мало что делают – это напрасная трата времени и основная проблема протокола HTTP. На отметке 120 мс, после того как браузер запросил первое изображение, ему *уже известно*, что понадобится второе изображение, но он ждет, когда соединение освободится, и вплоть до отметки 240 мс не может отправить запрос. Как мы увидим позже, существуют способы обойти это ограничение (например, большинство браузеров открывает сразу несколько соединений). Однако дело в том, что базовый протокол HTTP довольно неэффективен.

Большинство веб-сайтов состоят не только из двух изображений, и проблемы с производительностью возрастают с увеличением количества ресурсов, которые необходимо загрузить (как показано на рис. 2.2), – особенно это касается небольших ресурсов с небольшим объемом обработки с обеих сторон относительно сетевого запроса и времени отклика.

Одна из самых больших проблем современного Internet – это величина задержки, а не сама пропускная способность. *Задержка* показывает, сколько времени требуется, чтобы отправить одно сообщение на сервер, тогда как *пропускная способность* показывает максимальный объем информации, который пользователь может загрузить в сообщении. Новые технологии способствуют увеличению значения пропускной способности (что помогает решить проблему увеличения размера веб-сайтов), но величина задержки остается прежней (что не позволяет увеличивать количество запросов). Значение задержки ограничено постоянной фи-

зической величиной (скоростью света). Данные, передаваемые по оптоволоконным кабелям, уже идут со скоростью, близкой к скорости света, и, несмотря на развитие технологий, значительно повлиять на скорость данных невозможно.

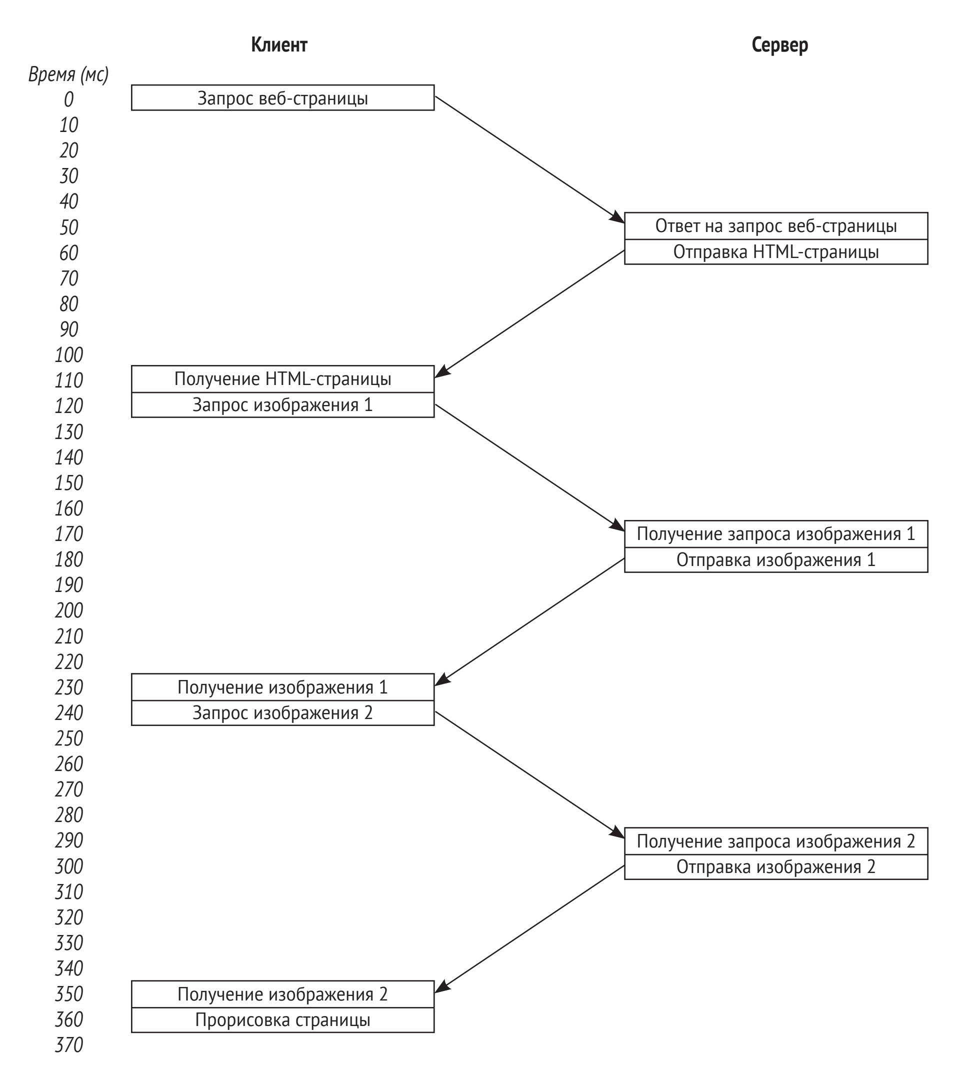


Рис. 2.2 Запросы и ответы по HTTP для базового примера веб-сайта

Майк Бэлш (Mike Belshe) из компании Google провел несколько экспериментов[1], согласно результатам которых мы достигли точки убываю-

[1] https://docs.google.com/a/chromium.org/viewer?a=v&pid=sites&srcid=Y2hyb21pdW0ub3JnfGRldnxneDoxMzcyOWI1N2I4YzI3NzE2.

щей отдачи от увеличения пропускной способности. Сегодня мы можем транслировать телевидение в высоком качестве, но процесс веб-серфинга не стал быстрее, и даже при высокоскоростном соединении на загрузку веб-сайтов часто требуется несколько секунд. Развитие Internet такими же большими темпами не представляется возможным без решения фундаментальных проблем производительности HTTP/1.1: на отправку и получение даже небольших HTTP-сообщений тратится слишком много времени.

2.1.2 Конвейеризация HTTP/1.1

Как было сказано в главе 1, разработчики HTTP/1.1 попытались реализовать *конвейерную обработку данных*, которая позволяет отправлять конкурентные запросы еще до получения ответов, благодаря чему становится возможной параллельная отправка запросов. Исходный код HTML по-прежнему нужно запрашивать отдельно, но, например, в случае, если браузеру потребуется загрузить два изображения, он сможет запросить их одно за другим.

Как вы можете увидеть на рис. 2.3, конвейеризация сокращает процесс на 100 мс, что в данном простом гипотетическом примере составляет треть всего времени обработки. Конвейеризация должна была значительно улучшить производительность HTTP, но по ряду причин ее было сложно реализовать, также присутствовал риск возникновения конфликтов и, кроме того, ее не поддерживали многие веб-браузеры и веб-серверы[1]. В результате использование конвейеризации – явление редкое. К примеру, эту технологию не использует ни один из самых известных веб-браузеров[2].

Но даже при условии, что многие серверы станут поддерживать конвейеризацию, ответы все равно должны возвращаться в том порядке, в котором они были запрошены. В случае, если изображение 2 доступно для скачивания, но изображение 1 еще не получено с другого сервера, то второе изображение не будет загружено, хотя технология и подразумевает возможность мгновенной передачи файлов. Эту проблему также называют *блокировкой очереди* (head-of-line blocking, HOL). Она часто встречается как в HTTP, так и в других сетевых протоколах. В главе 9 мы обсудим HOL-блокировку в TCP.

1 https://tools.ietf.org/html/draft-nottingham-http-pipeline-01#section-3.

2 https://en.wikipedia.org/wiki/HTTP_pipelining#Implementation_in_web_browsers.

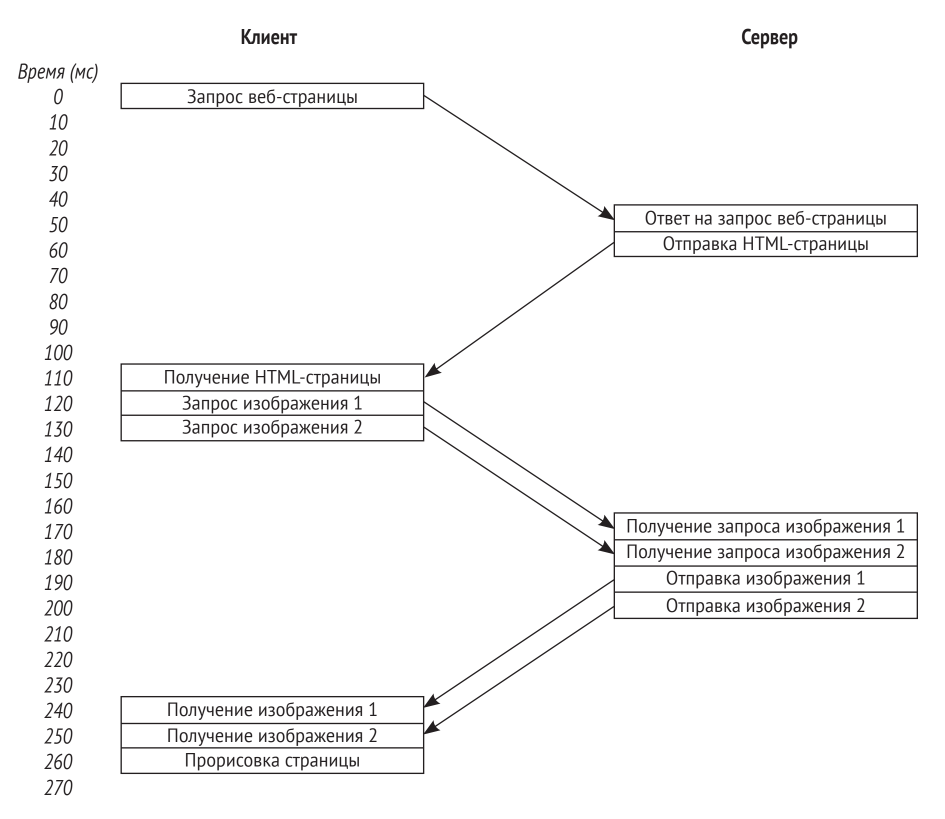


Рис. 2.3 Конвейеризация HTTP на примере базового веб-сайта

2.1.3 Использование каскадных диаграмм для анализа производительности

Потоки запросов и ответов (рис. 2.2 и 2.3) часто приводятся в виде каскадных диаграмм, где в колонке слева находятся активы (нужные файлы), а сверху, по направлению слева направо, – время. Такие диаграммы воспринимать легче, чем блок-схемы (рис. 2.2 и 2.3), используемые при наличии большего количества ресурсов. На рис. 2.4 показана каскадная диаграмма для нашего гипотетического сайта, а на рис. 2.5 показан тот же сайт, но уже с использованием конвейерной обработки.

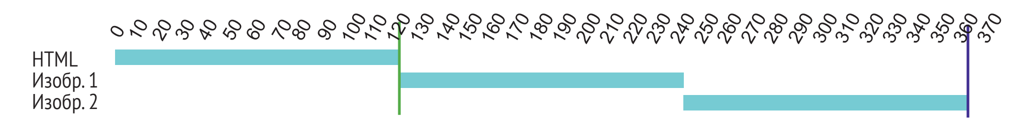


Рис. 2.4 Каскадная диаграмма гипотетического сайта

На обеих диаграммах первая вертикальная линия показывает время, когда отображается начальная страница (так называемое начало отрисовки, или начало рендеринга), а вторая вертикальная линия показывает

время, к которому страница полностью загружена. Очень часто браузеры пытаются отобразить страницу до того, как загрузятся изображения, поэтому последние часто загружаются между упомянутыми метками времени. Приведенные примеры довольно просты, однако существуют и более сложные случаи, которые мы рассмотрим в этой главе на практических примерах.

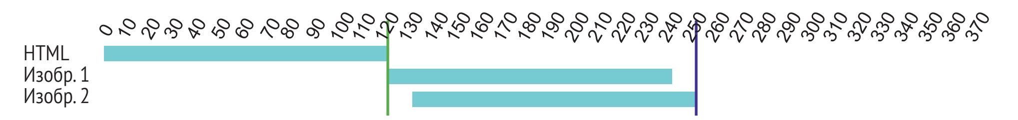


Рис. 2.5 **Каскадная диаграмма гипотетического сайта с использованием конвейеризации**

Существуют различные инструменты, такие как Webpagetest[1] и инструменты разработчика в веб-браузере (кратко описанные в главе 1), создающие каскадные диаграммы, которые в свою очередь очень важны при анализе производительности веб-сайтов. Большинство из таких инструментов разбивает общее время для каждого ресурса на компоненты (DNS и время соединения TCP), как показано на рис. 2.6.

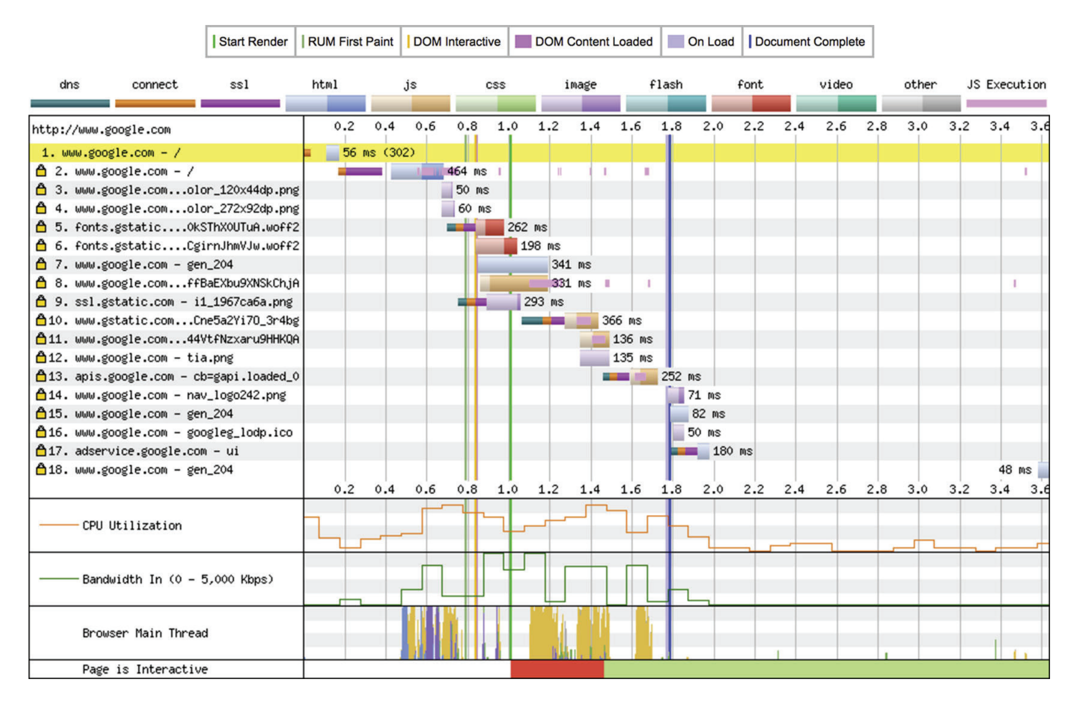


Рис. 2.6 **Каскадная диаграмма, созданная с помощью webpagetest.org**

---

[1] https://www.webpagetest.org.

В диаграмме на рис. 2.6 содержится гораздо больше информации, чем в простых каскадных диаграммах. Каждый запрос делится на несколько частей:

- DNS-поиск;
- время подключения к сети;
- время согласования HTTPS (или SSL);
- запрошенный ресурс (загрузка ресурса разделяется на две части: светлым оттенком обозначена загрузка запроса, а темным – загрузка ответа);
- вертикальные линии для разных этапов загрузки страницы;
- другие графы с информацией об использовании процессора, пропускной способности сети и указанием того, над чем в определенный работает основной поток браузера.

Вся эта информация полезна при анализе производительности веб-сайта. В нашей книге я часто привожу каскадные диаграммы при объяснении некоторых концепций.

2.2 Пути решения проблем с производительностью

Как было сказано ранее, HTTP/1.1 считается устаревшей версией протокола, потому что для получения ответа на один запрос он блокирует отправку других запросов. По сути, это синхронная работа; вы не можете отправить запрос, если предыдущий не был завершен. При медленном соединении HTTP работает хуже. Поскольку HTTP предназначен в первую очередь для запроса ресурсов с сервера, который зачастую находится далеко от клиента, низкая скорость сети превращается для протокола в суровую повседневную реалию. При использовании первоначальных вариантов HTTP (один HTML-документ) такая проблема не казалась весомым поводом для беспокойства. Однако, по мере того как веб-страницы становились сложнее и начинали требовать все больше и больше ресурсов, эта проблема встала в полный рост.

Поиск решений для медленных веб-сайтов породил отдельную область *оптимизации веб-производительности*. Кроме того, на эту тему было опубликовано множество книг и руководств. Решение проблем HTTP/1.1 было не единственным направлением оптимизации веб-производительности, однако большая часть области специализировалась именно на этом аспекте. Со временем начали появляться различные советы, хитрости и подсказки для преодоления ограничений производительности HTTP/1.1. Обычно у пользователя есть два пути:

- создание параллельных HTTP-соединений;
- создание меньшего количества HTTP-запросов, но теоретически большего размера.

Другие методы повышения производительности в меньшей степени связаны с протоколом HTTP. Они обеспечивают оптимальный запрос

ресурсов пользователем (например, сначала запрос критически важных элементов CSS), уменьшение объема загружаемых данных (сжатие изображений и адаптивные изображения) и эффективную работу браузера (более эффективный CSS или JavaScript). Такие методы по большей части выходят за рамки нашей книги, хотя я вернусь к некоторым из них в главе 6. Книга издательства Manning *Web Performance in Action* авторства Джереми Вагнера (Jeremy Wagner)[1] – отличный ресурс для более подробного изучения вышеописанных методов.

2.2.1 Создание параллельных HTTP-соединений

Один из самых простых способов решения проблемы блокировки HTTP/ 1.1 – использование нескольких соединений; благодаря распараллеливанию активными могут быть сразу несколько HTTP-запросов. Кроме того, каждое HTTP-соединение будет функционировать независимо от других, что поможет обойти блокировку HOL, которую не позволяет обойти конвейеризация соединений. Именно поэтому большинство браузеров открывают шесть соединений на один домен.

Чтобы обойти ограничение на шесть соединений, многие веб-сайты размещают статические ресурсы, такие как изображения, CSS и JavaScript на поддоменах (например, static.example.com), что позволяет открывать еще шесть подключений для каждого нового домена. Такой метод называется *доменным разделением* (не путать с сегментированием базы данных, хотя цели данных методов схожи). Доменное разделение, помимо увеличения количества параллельных соединений, несет в себе и другие полезные функции, например сжатие HTTP-заголовков (таких как файлы куки (см. раздел 2.3)). Зачастую разделенные домены размещаются на одном сервере. Распределение ресурса по разным доменным именам вводит браузер в заблуждение, и он видит уже несколько разных серверов. На рис. 2.7 приведен пример того, как использует доменное разделение сайт stackoverflow.com: JQuery загружается из домена Google, скрипты и таблицы стилей из cdn.static.net и изображения из i.stack.imgur.com.

Использование параллельных HTTP-соединений кажется простым решением, однако это не совсем так, и у этого метода есть некоторые недостатки. Например, появляются дополнительные расходы как для клиента, так и для сервера: запуск TCP-соединения требует времени, а поддержание соединения требует затрат памяти и увеличивает объем обработки.

Однако основная проблема параллельных HTTP-соединений заключается в неэффективности лежащего в основе протокола TCP. Этот протокол обеспечивает надежную доставку данных, где каждый пакет имеет уникальный порядковый номер, а также повторно запрашивает любые пакеты, которые были потеряны в пути, выявляя отсутствующие порядковые номера. Для установления TCP-соединения требуется трехстороннее рукопожатие, как показано на рис. 2.8.

---

[1] https://www.manning.com/books/web-performance-in-action.

Name	Status	Domain	Type
stackoverflow.com	200	stackoverflow.com	document
jquery.min.js ajax.googleapis.com/ajax/libs/jquery/1.12.4	200	ajax.googleapis.com	script
stub.en.js?v=1ec6f067df10 cdn.sstatic.net/Js	200	cdn.sstatic.net	script
stacks.css?v=4fe27c331a7b cdn.sstatic.net/Shared	200	cdn.sstatic.net	stylesheet
primary-unified.css?v=92dbb274d371 cdn.sstatic.net/Sites/stackoverflow	200	cdn.sstatic.net	stylesheet
yQoqq.png?s=48&g=1 i.stack.imgur.com	200	i.stack.imgur.com	png
6HFc3.png i.stack.imgur.com	200	i.stack.imgur.com	png
5d55j.png	200	i.stack.imgur.com	png
vobok.png i.stack.imgur.com	200	i.stack.imgur.com	png

Рис. 2.7 Доменное разделение stackoverflow.com

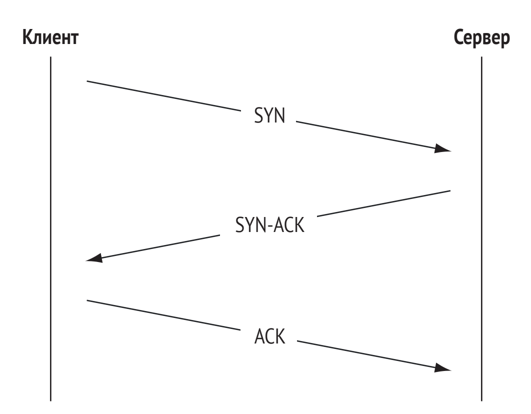


Рис. 2.8 Трехстороннее рукопожатие для TCP

Рассмотрим данные шаги подробнее.

1. Клиент отправляет сообщение синхронизации (SYN) с порядковым номером пакета, на котором будут основываться номера всех последующих TCP-пакетов из этого запроса.
2. Сервер подтверждает полученные порядковые номера и отправляет клиенту ответный запрос синхронизации, где прописаны порядковые номера, которые он будет использовать. Оба вышеописанных сообщения объединяются в одно сообщение SYN-ACK.
3. Затем клиент подтверждает информацию от сервера посредством отправки сообщения ACK.

Чтобы отправить всего один HTTP-запрос, необходимо совершить три цикла приема и передачи информации (или 1,5 двусторонних цикла)!

Кроме того, запуск TCP происходит очень осторожно, и до подтверждения отправляется лишь небольшое количество пакетов. С течением времени *окно перегрузки* (CWND) постепенно увеличивается, так как уже становится ясно, что соединение может обрабатывать большие объемы информации и не терять при этом пакеты. Окно перегрузки контролирует *алгоритм замедленного старта* TCP. Ввиду того, что TCP призван обеспечить надежное сетевое соединение, его задача – не допустить перегрузку сети. Прежде чем можно будет отправить больше пакетов с использованием приращений порядковых номеров, установленных в трехстороннем рукопожатии, пакеты TCP должны пройти проверку алгоритма замедленного старта. Если размер окна перегрузки небольшой, может потребоваться несколько подтверждений TCP для отправки полных сообщений HTTP-запроса. HTTP-ответы, которые зачастую намного объемнее, чем HTTP-запросы, также подвержены ограничениям окна перегрузки. По мере увеличения частоты использования TCP-соединения увеличивается и окно перегрузки, что обеспечивает более эффективную работу. Однако даже при высокоскоростном соединении с высокой пропускной способностью оно всегда искусственно ограничивается при запуске. В главе 9 мы вернемся к этой теме, но уже и сейчас понятно, что параллельные TCP-соединения не обходятся даром.

Кроме того, проблемы с пропускной способностью могут возникнуть и при использовании замедленного старта нескольких параллельных независимых соединений. Например, в результате использования одним соединением максимально возможной пропускной способности могут возникать паузы в работе TCP и повторные передачи для других соединений. Концепция установления приоритетов между трафиком независимых подключений для наиболее эффективного использования доступной пропускной способности еще не создана.

После установки TCP-соединения многие сайты, заботящиеся о безопасности, требуют установить HTTPS-соединение. При последующих соединениях многие параметры основного HTTPS-соединения можно использовать повторно, вместо того чтобы устанавливать каждое соединение с нуля, однако этот процесс по-прежнему требует осуществления дополнительных циклов передачи данных и, следовательно, дополнительных затрат времени. В главе 4 мы рассмотрим HTTPS-рукопожатия более подробно.

Можно сделать вывод, что использование параллельных TCP- и HTTPS-соединений неэффективно, несмотря на то что они эффективны на уровне HTTP. Для HTTP/1.1 решение проблем с задержкой требует осуществления нескольких дополнительных циклов запросов и ответов; следовательно, решение создает те же проблемы, которые оно призвано решить!

Помимо этого, к тому времени, когда TCP-соединение достигнет максимальной производительности, вполне вероятно, что большая часть веб-страницы будет уже загружена и дополнительные соединения больше не потребуются. Даже переход на последующие страницы может обойтись без загрузки существенных ресурсов, если общие элементы

кешированы. *Патрик Макманус* (Patrick McManus) из *Mozilla* утверждает, что при использовании HTTP/1.1 74 % активных подключений выполняют в Mozila только одну транзакцию. Позже мы рассмотрим несколько практических примеров на эту тему.

Таким образом, параллельные TCP-соединения являются не самым лучшим путем решения проблем HTTP/1, хотя если у вас нет других вариантов, то можно использовать и их. К слову, это объясняет, почему браузеры ограничивают количество соединений для одного домена до шести. Однако возможно установить и больше соединений (как в некоторых браузерах), но тогда отдача будет сокращаться в связи с ростом расходов, необходимых для каждого соединения.

2.2.2 Сокращение количества запросов

Другой распространенный метод оптимизации – сокращение количества необязательных запросов (например, путем кеширования ресурсов в браузере) или передача определенного объема данных путем выполнения меньшего количества запросов. Первый метод предусматривает использование кеширования, которое кратко описывается в главе 1, а более подробно рассматривается в главе 6. Второй метод заключается в сборе ресурсов в комбинированные файлы.

Изображения группируются посредством *спрайтинга*. Если на вашем веб-сайте размещено несколько иконок социальных сетей, вы можете использовать отдельный файл для каждой из них. Такой метод, однако, приведет к появлению большого количества HTTP-очередей, что значительно снизит эффективность. Это происходит, потому что иконки представляют собой маленькие изображения, поэтому большая часть времени будет потрачена на их загрузку. Во избежание такого исхода вы можете объединить иконки в большой графический файл, а затем средствами CSS извлекать части этого файла. Таким образом, ваши изображения будут загружаться быстрее и эффективнее. На рис. 2.9 приведен пример вышеописанного графического файла, используемого в TinyPNG и содержащего все нужные иконки.

Что касается CSS и JavaScript, многие веб-сайты объединяют несколько файлов в один, благодаря чему файлов становится меньше, но при этом не уменьшается объем кода. Эта конкатенация часто сопровождается минимизацией CSS или JavaScript путем удаления неотображаемых знаков, комментариев и других ненужных элементов. Оба метода дают повышение производительности, но требуют усилий для их внедрения.

Существуют и другие методы. Например, встраивание ресурсов в другие файлы. Критически важный код CSS часто включается непосредственно в HTML, например с помощью тегов `<style>`. Также изображения могут быть встроены в CSS в виде инструкций *масштабируемой векторной графики* (Scalable Vector Graphic, SVG) или двоичных файлов в кодировке base64, что позволяет обойтись без дополнительных HTTP-запросов.

Основным недостатком такого метода является его сложность. Создание спрайт-файлов требует усилий; легче загружать изображения

в виде отдельных файлов. Не все веб-сайты используют этап сборки, на котором можно автоматизировать оптимизацию, например объединить файлы CSS. Если на своем сайте вы используете систему управления контентом, то, скорее всего, не можете автоматически объединять JavaScript или спрайт-файлы.

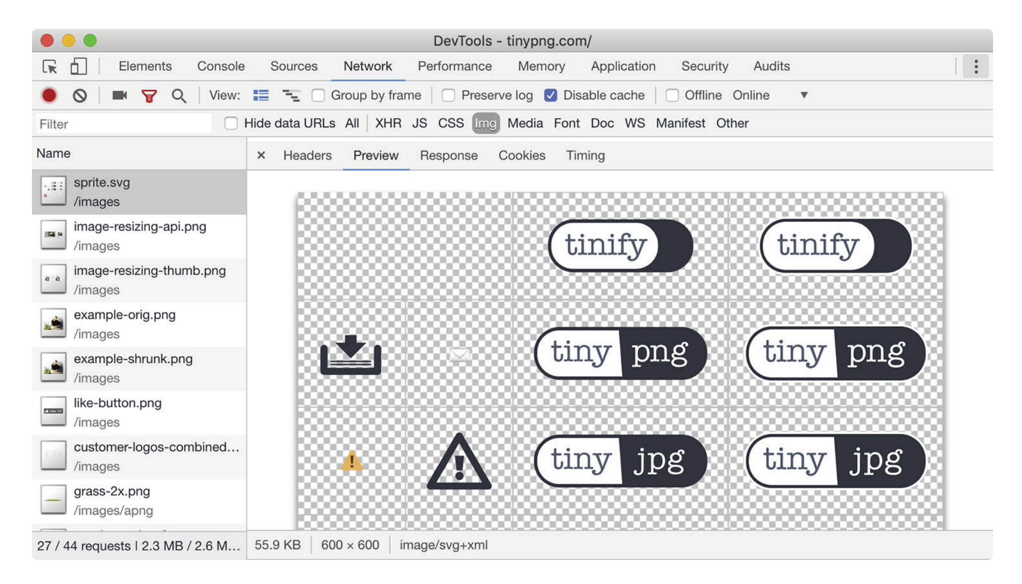


Рис. 2.9 Спрайт-файл для TinyPNG

Другой недостаток – избыточный расход трафика на эти файлы. Некоторые страницы загружают объемные спрайт-файлы, но используют только одно или два изображения. Отследить, какая часть спрайт-файла активна, а какая уже не используется, довольно сложно. Чтобы изображения корректно загружались из нужных мест в новый спрайт-файл, вам потребуется переписать весь CSS. Точно так же увеличивается объем кода JavaScript, если вы объедините слишком много элементов и загрузите объемный файл. Данный метод неэффективен как на сетевом уровне (особенно из-за медленного запуска TCP поначалу), так и с точки зрения обработки, поскольку веб-браузеру приходится обрабатывать данные, которые он не будет использовать.

И последний недостаток связан с кешированием. Если вы кешируете свои спрайт-файлы в течение длительного времени (чтобы посетители сайта не загружали их слишком часто), а затем захотите добавить изображение, браузеру придется загрузить весь файл заново, даже несмотря на то, что посетитель может не нуждаться в этом изображении. Вы можете использовать различные методы, такие как добавление номера версии в имя файла или использование параметра запроса[1], но все они также затратны. Аналогично малейшие изменения кода CSS или JavaScript повлекут за собой полную перезагрузку скомпонованного файла.

---

[1] https://css-tricks.com/strategies-for-cache-busting-css/.

2.2.3 Вывод

В конечном счете оптимизация производительности HTTP/1 – это лишь ухищрения в попытке обойти основные недостатки протокола HTTP. Однако было бы намного эффективнее исправить этот недостаток на уровне протокола, чтобы сэкономить время и силы. Именно для этой цели и служит версия HTTP/2.

2.3 Другие проблемы HTTP/1.1

HTTP/1.1 – это простой текстовый протокол, но из-за его простоты и возникает ряд проблем. Тела сообщений HTTP могут содержать двоичные данные (например, изображения в любом формате, который подходит клиенту и серверу), однако запросы и сами заголовки должны быть в текстовом виде. Текстовый формат отлично подходит для людей, но не подходит для машин. Обработка текстовых HTTP-сообщений – это сложный процесс, в ходе которого могут возникать ошибки, что порождает проблемы с безопасностью. Например, некоторые атаки на HTTP-серверы были совершены посредством ввода символов в HTTP-заголовки[1].

Существуют и другие проблемы, связанные с текстовым форматом протокола. Например, одна из них состоит в том, что сообщения HTTP больше, чем нужно на самом деле. Происходит это из-за неэффективного кодирования данных (например, дату в заголовке можно было бы представить в виде числа, а не полного текста, который понятен человеку) и повторяющихся заголовков. Опять же, в первоначальном варианте использования Internet (с отдельными запросами) такая ситуация не представляла большой проблемы, но растущее количество запросов выводит эту проблему на первый план. Сегодня используется большое количество HTTP-заголовков, многие из которых повторяются. Файлы куки, например, отправляются с каждым HTTP-запросом в домен, даже если эти файлы требуются только для запроса главной страницы. Обычно статические ресурсы, такие как изображения, CSS и JavaScript, не нуждаются в файлах куки. Доменное разделение, как сказано ранее в этой главе, было придумано, чтобы обеспечить возможность открывать дополнительные подключения, но также используется для создания специальных доменов без файлов куки, которым не нужно отправлять файлы куки из соображений производительности и безопасности. Количество HTTP-ответов также растет, и при использовании HTTP-заголовков, созданных для безопасности (например, Content-Security-Policy), недостатки текстового протокола становятся более очевидными, так как их объемы велики. Поскольку многие веб-сайты состоят из 100 и более ресурсов, большие HTTP-заголовки могут добавить десятки или сотни килобайт передаваемых данных.

1 https://www.owasp.org/index.php/Testing_for_HTTP_Splitting/Smuggling_(OTG-INPVAL-016).

Ограничения производительности – это лишь один аспект HTTP/1.1, который нуждается в улучшении. Среди других проблем – безопасность и конфиденциальность протокола открытого текста (которые довольно успешно решаются с помощью HTTPS) и отсутствие записи состояния (в какой-то мере решаются с помощью файлов куки). В главе 10 мы подробнее рассмотрим эти вопросы. Многие считают, что проблемы с производительностью решить довольно тяжело, не применяя обходные пути, которые в свою очередь создают новые проблемы.

2.4 Практические примеры

Как мы уже знаем, HTTP/1.1 неэффективен при использовании множественных запросов. Но насколько эта проблема серьезна? Заметна ли она? Давайте рассмотрим несколько практических примеров.

Реальные веб-сайты и HTTP/2

Когда я начал писать эту главу, оба веб-сайта, которые я использовал для примеров, не поддерживали HTTP/2. Сейчас они его поддерживают, но примеры, показанные здесь, по-прежнему актуальны, поскольку представляют сложные веб-сайты, работающие неэффективно при использовании HTTP/1.1, и многие из обсуждаемых здесь деталей, вероятно, аналогичны таковым для других веб-сайтов. HTTP/2 набирает популярность, и любой сайт, выбранный в качестве примера, могут обновить. Я предпочитаю демонстрировать проблемы, которые могут быть решены с помощью HTTP/2 на примерах реальных, широко известных сайтов. Я не использую искусственные примеры веб-сайтов, созданных исключительно для подтверждения моей точки зрения, поэтому я сохранил два исходных примера веб-сайтов, несмотря на то что теперь они поддерживают HTTP/2. Для нас больше важны концепции, которые они демонстрируют.

Чтобы повторить эти тесты на webpagetest.org, вы можете отключить HTTP/2, указав опцию `--disable-http2` (**Дополнительные настройки** > **Chrome** > **Параметры командной строки**). В Firefox есть аналогичные опции[a]. Кроме того, вы можете тестировать производительность и на своих собственных веб-сайтах.

[a] https://www.webpagetest.org/forums/showthread.php?tid=14162.

2.4.1 Пример 1: amazon.com

Если до этого мы разбирали теорию, то сейчас пришло время перейти к практике. Если сделать тест www.amazon.com на www.webpagetest.org, получится каскадная диаграмма как на рис. 2.10. На нем мы можем увидеть многие проблемы HTTP/1.1.

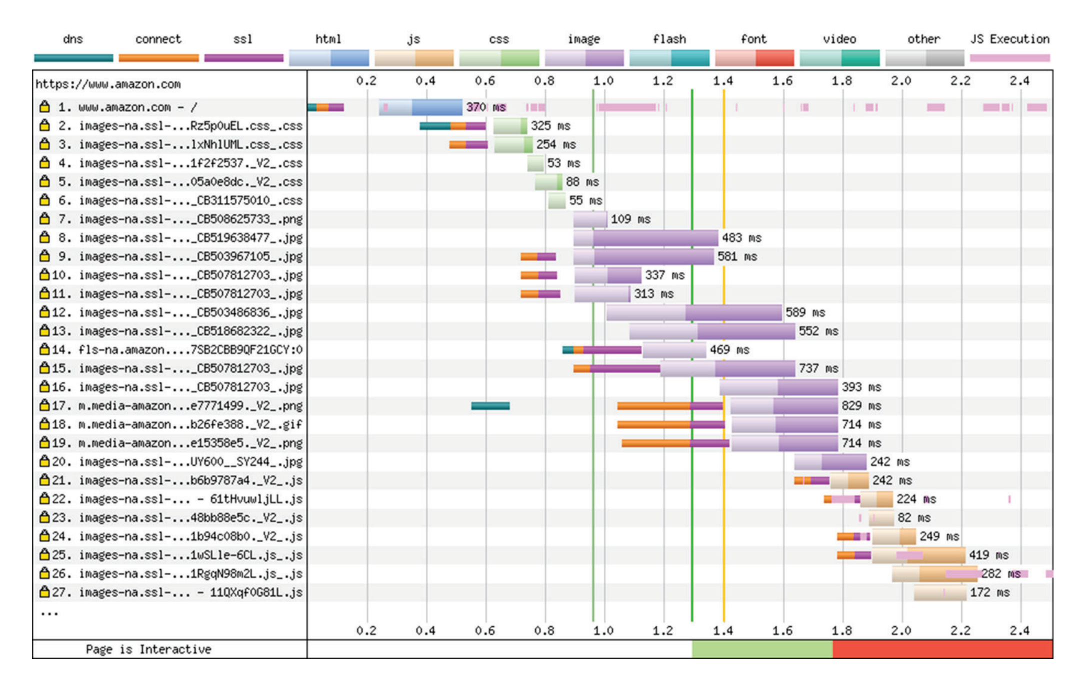


Рис. 2.10 Фрагмент результатов для amazon.com

- Первый запрос – запрос домашней страницы. На рис. 2.11 мы можем рассмотреть его ближе.

  Отправка одного запроса, поиска DNS, запуск подключения и согласование SSL/TLS HTTPS – все эти действия отнимают время. По отдельности процессы занимают не так уж и много времени (чуть больше 0,1 с, согласно рис. 2.11), но, когда затраченное время складывается, задержка становится ощутимой. И при выполнении первого запроса с этим нечего не поделаешь. Как было сказано в главе 1, это неотъемлемая часть того, как работает Internet. Вы можете оптимизировать HTTPS и сократить задержку при согласовании с SSL, но это в любом случае не коснется первого запроса. Лучшее, что вы можете сделать, – это убедиться в том, что ваши серверы быстро реагируют на запросы, и – в идеале – находятся рядом с пользователями. Таким образом, вы сможете сократить время приема–передачи настолько, насколько это возможно. В следующей главе мы обсудим *сети доставки контента* (Content Delivery Network, CDN), которые помогают решить вышеописанную проблему.

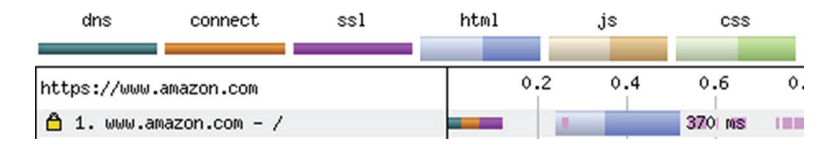


Рис. 2.11 Первый запрос домашней страницы

После запуска на диаграмме мы видим небольшую паузу, вероятно, произошедшую из-за неточного тайминга или проблем браузера Chrome (при выполнении теста в браузере Firefox пауза отсутствовала). Далее выполняется первый HTTP-запрос (процесс обозначен светлым оттенком), а затем загружается, анализируется и обрабатывается HTML-код (процесс обозначен оттенком темнее).

- HTML содержит ссылки на несколько загружаемых CSS-файлов, как показано на рис. 2.12.

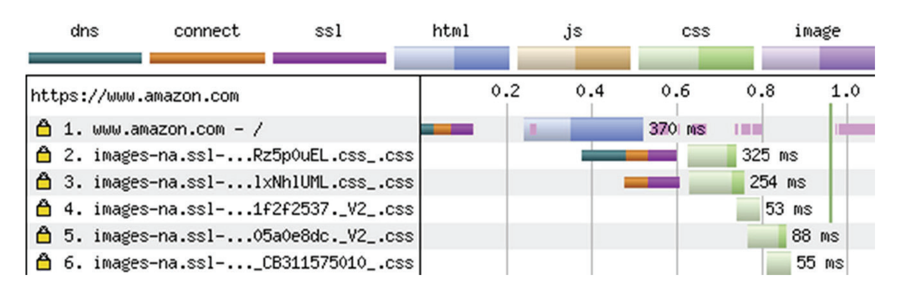


Рис. 2.12 Пять запросов CSS-файлов

- Данные файлы размещены на другом домене (images-na.ssl-imagesamazon.com), который был отделен от основного домена для улучшения производительности. В связи с этим, прежде чем использовать этот домен для загрузки CSS, для второго запроса вам следует выполнить поиск нового DNS, использовать новое сетевое соединение и заново согласовать HTTPS. Доменное разделение применяется для решения проблем с производительностью, и если задержка для настройки первого запроса неизбежна, то время на организацию второго запроса тратится напрасно. Также обратите внимание, что CSS-файл появляется еще на ранней стадии обработки HTML-страницы первого запроса, в результате чего второй запрос начинается немного раньше отметки 0,4 с, несмотря на то что HTML-страница загружается не раньше чем через 0,5 с. Браузер не дождался загрузки и полной обработки HTML-страницы; вместо этого, как только он заметил указанный домен, он запросил дополнительное HTTP-соединение (несмотря на то что сам ресурс не начал загружаться до тех пор, пока браузер не получил весь HTML-код ввиду задержки установки соединения).
- Затем идет запрос еще одного CSS-файла в том же отделенном домене. Ввиду того, что HTTP/1.1 не позволяет выполнять два запроса одновременно, браузер создает еще одно соединение. Теперь вам не нужно искать DNS, потому что вы уже знаете IP-адрес данного домена из запроса 2. Однако, прежде чем запросить CSS, вам все же нужно установить TCP/IP-соединение и согласовать HTTPS, на что снова тратится много времени. Напомню, что дополнительное соединение создается в целях решения проблем с производительностью HTTP/1.1.

- Затем браузер запросит еще три CSS-файла, которые загрузятся через два уже установленных соединения. На диаграмме не видно, почему браузер не запросил эти файлы сразу – для этого пришлось бы создать еще несколько дополнительных соединений, что привело бы к соответствующим затратам. При просмотре исходного кода страниц Amazon выяснилось, что перед запросом данных файлов стоит тег `<script>`, который блокирует последующие запросы, до тех пор пока сценарий не будет полностью обработан. Именно поэтому запросы 4, 5 и 6 не могут выполняться одновременно с запросами 2 и 3. К этой теме мы вернемся несколько позже. Несмотря на то что неэффективность HTTP/1.1 является серьезной проблемой и решается она путем улучшений протокола HTTP (например, HTTP/2), это далеко не единственная причина низкой производительности веба.
- После обработки CSS-файлов из запросов 2–6 загружаются изображения. Браузер начинает их загрузку, как показано на рис. 2.13.

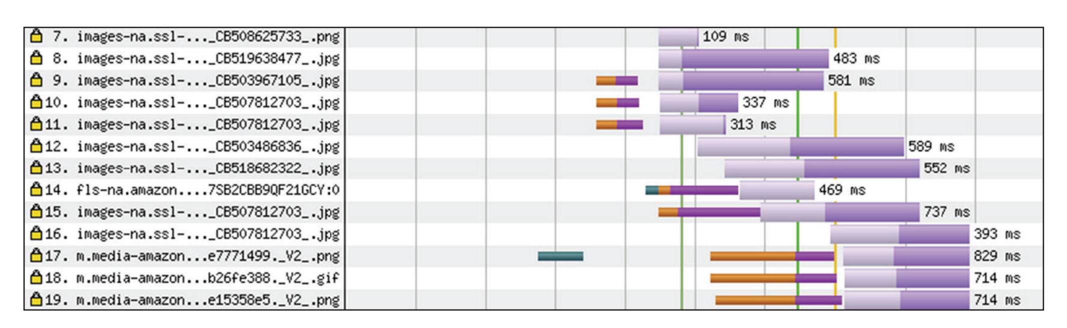


Рис. 2.13 Загрузка изображений

- Первый файл .png в запросе 7 представляет собой спрайт-файл, содержащий несколько изображений (как на рис. 2.13) и реализующий нескольких других хитростей по повышению производительности от Amazon. В следующих запросах загружается несколько файлов формата .jpg.
- В процессе осуществления запросов изображений браузеру приходится установить более ресурсоемкие соединения. Это нужно для того, чтобы другие файлы загружались параллельно в запросах 9, 10, 11 и 15 и в запросах 14, 17, 18 и 19 для новых доменов.
- В некоторых случаях (в запросах 9, 10 и 11) браузер предвидел, что потребуется больше соединений, и настроил их заранее. Именно поэтому установка соединения и настройка SSL выполняются раньше, ввиду чего браузер может запрашивать изображения в запросах 7 и 8.
- Amazon добавил в оптимизацию производительности предвыборку DNS[1] для m.mediaamazon.com. Как ни странно, для fls-na.amazon.com этого не сделали. По этой причине в запросе 17 поиск DNS про-

[1] https://css-tricks.com/prefetching-preloading-prebrowsing/.

исходит на отметке 0,6 с, т. е. заранее. В главе 6 мы поговорим об этом поподробнее.

После выполнения всех вышеперечисленных запросов загрузка все еще продолжается, но даже на примере первых нескольких запросов видно, какие проблемы существуют у HTTP/1.1.

Организация большого количества соединений нужна для предотвращения очередей. Время, затраченное на это, очень часто удваивает время, необходимое для загрузки ресурса. WebPageTest организует данные соединений в удобном для восприятия формате[1] (как показано на рис. 2.14).

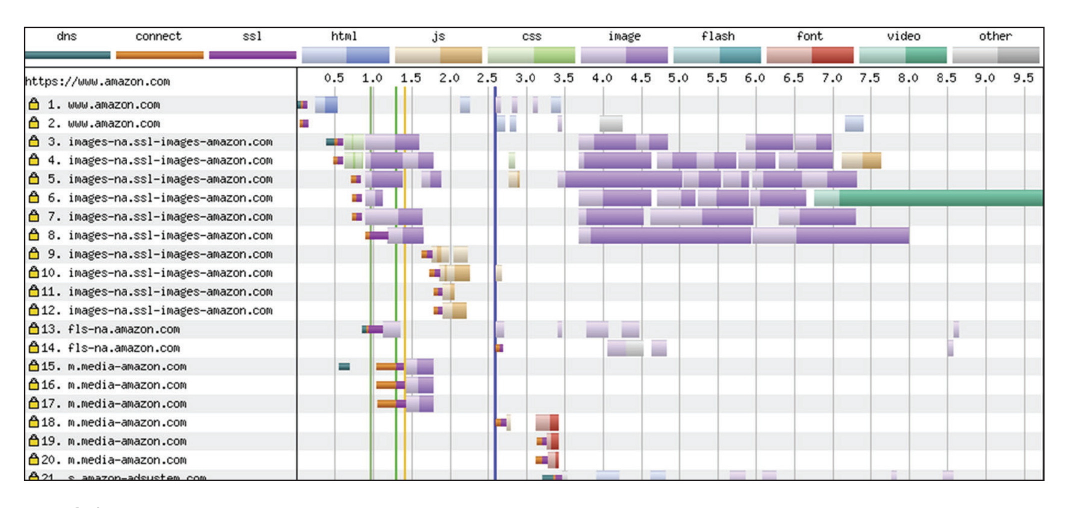


Рис. 2.14 Визуализация загрузки amazon.com

Как видите, для загрузки основного контента amazon.com приходится установить 20 подключений, а для загрузки рекламных ресурсов к ним добавляется еще 28 (процесс не показан на рис. 2.14). С помощью первых 6 соединений `images-na.sslimages-amazon.com` загружается большое количество ресурсов (соединения 3–8), а остальные соединения (соединения 9–12, а также 15–20) загружают по одному или два ресурса. Таким образом, затраты большого количества времени на установку последних соединений нерациональны.

Причина, по которой `images-na.sslimages-amazon.com` устанавливает эти соединения (и по которой Chrome зачастую устанавливает более 6 подключений на домен), довольно интересна и требует небольшого исследования. Запросы можно отправлять как с *идентификационными данными* (обычно это файлы куки), так и без них (тогда Chrome обработает их через отдельные соединения). По соображениям безопасности, ввиду того, что запросы из разных источников обрабатываются в браузерах

[1] https://www.webpagetest.org/result/170820_NR_53c5bf9ca1e67301a933947d80a32a53/1/details/#connectionView_fv_1.

по-разному[1], при запросах файлов JavaScript, не содержащих идентификационных данных, Amazon использует метод `setAttribute("crossorigin", "anonymous")`. Существующие соединения уже не используются, а вместо них создается больше новых соединений. Для прямых запросов JavaScript, содержащих тег `<script>` в HTML-коде, использование такого метода не требуется. Также он не требуется при загрузке ресурсов, размещенных в основном домене, что еще раз показывает, что доменное разделение на уровне HTTP не всегда эффективно.

На примере Amazon мы видим, что использование обходных путей все-таки снижает производительность HTTP1/1. К тому же их довольно сложно реализовать. От сайтов требуется, чтобы они могли управлять несколькими доменами и спрайт-файлами или объединять весь JavaScript (или CSS) в один файл. Не каждый сайт способен на это, и не у каждого сайта есть ресурсы оптимизации, как у Amazon. Небольшие сайты зачастую менее оптимизированы и больше страдают от ограничений HTTP/1.

2.4.2 Пример 2: imgur.com

Что будет, если не использовать оптимизацию? В качестве примера приведем сайт imgur.com. Поскольку этот сайт предназначен для обмена изображениями, он загружает огромное количество изображений на главную страницу, но не группирует их в спрайт-файлы. На рис. 2.15 показана часть каскадной диаграммы данного сайта от WebPagetest.

Первая часть загрузки страницы (до запроса 31) во многом похожа на загрузку страницы amazon.com, поэтому ее мы пропустим. На рисунке мы можем видеть, что для загрузки запросов 31–36 используется 6 соединений, а остальные поставлены в очередь. По завершении каждого из данных запросов могут быть запущены еще 6 новых, за которыми может последовать еще столько же. За счет этого и образуется характерная каскадная форма, за счет которой такие диаграммы и получили свое название. Обратите внимание, что ресурсы не связаны между собой, а их загрузка может завершиться в разное время (как это и происходит на нашей диаграмме). Однако, если их размер совпадает, их загрузка нередко заканчивается в одно и то же время. Таким образом, возникает иллюзия того, что ресурсы связаны, но на уровне HTTP это не так (хотя пропускная способность у них одинакова, а также они обращаются к одному серверу).

На рис. 2.16 изображена каскадная диаграмма для Chrome. Здесь проблема становится более очевидной, поскольку на диаграмме видна длительность задержки с момента, когда уже можно было бы отправить запрос ресурса. Как видите, у более поздних запросов задержка перед запросом изображения увеличивается (выделено прямоугольником), а время загрузки становится относительно короче.

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

dns | connect | ssl | html | js | css | image | flash | font | video | other | JS Execution

http://imgur.com/

Запрос	Время
...	
31. i.imgur.com – ISQ4VRDb.jpg	487 ms
32. i.imgur.com – 1fP5pk8b.jpg	482 ms
33. i.imgur.com – tj2tAnsb.jpg	480 ms
34. i.imgur.com – pkvTdRwb.jpg	480 ms
35. i.imgur.com – tFwWdLGb.jpg	477 ms
36. i.imgur.com – 6wygCgXb.jpg	474 ms
37. i.imgur.com – U6qPHbnb.jpg	210 ms
38. i.imgur.com – w0SzyZQb.jpg	207 ms
39. i.imgur.com – 6CjRtJQb.jpg	209 ms
40. i.imgur.com – bDuq5FFb.jpg	210 ms
41. i.imgur.com – hB9vLS0b.jpg	200 ms
42. i.imgur.com – RfrkiLhb.jpg	199 ms
43. i.imgur.com – 2DobzGob.jpg	107 ms
44. i.imgur.com – Cvq0ukhb.jpg	99 ms
45. i.imgur.com – PcXvWDmb.jpg	45 ms
46. i.imgur.com – 0MwEJZSb.jpg	87 ms
47. i.imgur.com – Wi0i2gmb.jpg	97 ms
48. i.imgur.com – EpDIHA4b.jpg	103 ms
49. i.imgur.com – K53I1Fbb.jpg	66 ms
50. i.imgur.com – iytwM0pb.jpg	40 ms
51. i.imgur.com – ALmNYq7b.jpg	37 ms
52. i.imgur.com – fip6mUJb.jpg	75 ms
53. i.imgur.com – qvjznB8b.jpg	68 ms
54. i.imgur.com – Epyxv18b.jpg	63 ms
55. i.imgur.com – sHmg4EXb.jpg	66 ms
56. i.imgur.com – aNe7TdVb.jpg	69 ms
57. i.imgur.com – j5MYqJqb.jpg	66 ms
58. i.imgur.com – R1qpg3Ub.jpg	59 ms
59. i.imgur.com – 7LQU2RXb.jpg	49 ms
60. i.imgur.com – wt3vhbKb.jpg	83 ms
...	

0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5

Page is Interactive

Рис. 2.15 Часть каскадной диаграммы сайта imgur.com

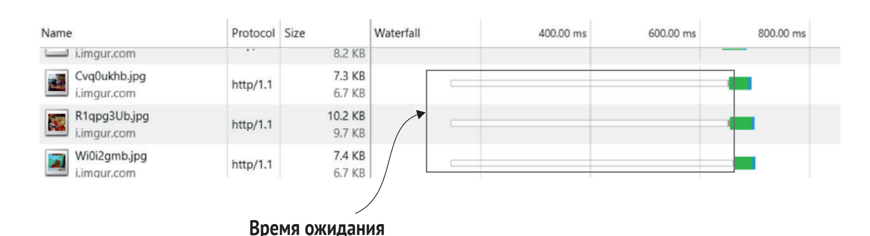


Рис. 2.16 Каскадная диаграмма в панели инструментов разработчика Chrome для сайта imgur.com

2.4.3 Насколько проблема серьезна?

Несмотря на недостатки HTTP, существуют способы их обойти. Однако они требуют времени, денег и достаточного уровня знаний для реализации и поддержки в будущем, а также у них есть собственные проблемы с производительностью. Заставлять разработчиков тратить время на работу с неэффективным протоколом довольно дорого (не говоря уже о том, что многие владельцы сайтов не осознают влияние низкой производительности на трафик). Многочисленные исследования показыва-

ют, что пользователи отказываются от использования медленных веб-сайтов, что приводит к снижению продаж[1,2].

Также необходимо понимать, насколько серьезна данная проблема по сравнению с другими проблемами производительности. Существует множество причин, по которым веб-сайты работают медленно: качество подключения, размер веб-сайта, избыток кода JavaScript, а также перегрузка неоптимизированной рекламой и средствами отслеживания поведения пользователей. Но все же невозможность быстрой и эффективной загрузки ресурсов является одной из главных составляющих проблемы. Однако, несмотря на это, многие веб-сайты все еще не оптимизированы. Очевидно, что многие владельцы сайтов обеспокоены этим, поэтому они прибегают к обходным путям, но для некоторых это слишком сложный способ.

Еще одна проблема состоит в том, что обходные пути имеют собственные ограничения, вследствие чего становятся неэффективными. Кроме того, поскольку веб-сайты продолжают расти в размерах и становятся все сложнее, в какой-то момент эти обходные пути вовсе перестанут работать. Несмотря на то что браузеры способны открывать шесть соединений на один домен и даже могут увеличить это число, расходы на это, по сравнению с достижением положительного результата, достигают точки убывающей отдачи. Именно поэтому браузеры и ограничивают количество подключений шестью, даже несмотря на то, что многие владельцы сайтов пытались обойти это ограничение с помощью доменного разделения.

Так или иначе, каждый веб-сайт индивидуален, и каждый владелец веб-сайта или веб-разработчик должен уделить должное время поиску узких мест в своих ресурсах, и в этом им могут помочь такие инструменты, как каскадные диаграммы, на которых видно, насколько сильно недостатки HTTP/1.1 влияют на производительность.

2.5 Переход от HTTP/1.1 к HTTP/2

С момента выхода версии HTTP/1.1 в 1999 году HTTP практически не изменялся. В 2014 году был опубликован новый Request for Comments (RFC), в котором уточнялась спецификация протокола. Однако каких-либо изменений в нем не произошло, а документ был создан для того, чтобы официально задокументировать протокол. Также какое-то время велась работа над обновленной версией (HTTP-NG), которая должна была кардинально поменять работу HTTP. В 1999 году от этой разработки отказались, так как многие считали такие изменения не применимыми к внедрению в реальную практику.

1 https://developers.google.com/web/fundamentals/performance/why-performance-matters/.

2 https://developer.akamai.com/blog/2016/09/14/mobile-load-time-user-abandonment.

2.5.1 SPDY

В 2009 году Майк Бэлш (Mike Belshe) и Роберт Пеон (Robert Peon) из Google объявили, что работают над новым протоколом под названием SPDY (название не является акронимом и произносится как speedy). Они экспериментировали с этим протоколом в лабораторных условиях и получили отличные результаты – скорость загрузки страницы увеличивалась до 64 %. Эксперименты проводились на копиях 25 реальных веб-сайтов, а не на гипотетических веб-сайтах.

SPDY создан на основе HTTP, и при этом не внес в протокол кардинальных изменений, подобно HTTPS. HTTP-методы (`GET`, `POST` и т. д.) и концепция HTTP-заголовков в SDPY остались неизменными. SPDY работает на нижнем уровне, поэтому практически прозрачен для веб-разработчиков, владельцев серверов и, что особенно важно, для пользователей. Любой HTTP-запрос просто преобразовывается в запрос SPDY, отправляется на сервер, а затем конвертируется обратно. Этот запрос выглядит как любой другой HTTP-запрос к приложениям более высокого уровня (таким как приложения JavaScript на стороне клиента). Кроме того, SPDY работает только по защищенному протоколу HTTP (HTTPS), что позволяет скрыть структуру и формат сообщения от других пользователей сети Internet. Все существующие сети, маршрутизаторы, коммутаторы и другие элементы инфраструктуры обрабатывают SPDY-сообщения точно так же, как и сообщения HTTP/1. SPDY обладает обратной совместимостью, и его внедрение не требует грандиозных изменений и снижает риски. Именно поэтому SPDY добился успеха, чего мы не можем сказать о HTTP-NG. И если HTTP-NG пытался решить многочисленные проблемы HTTP/1, то основной целью SPDY было устранение ограничений производительности HTTP/1.1. В нем представлено несколько важных концепций, таких как:

- *мультиплексированные потоки* (запросы и ответы использовали одно TCP-соединение и разбивались на чередующиеся пакеты, сгруппированные в отдельные потоки);
- *приоритизация запросов* (позволила избежать появления новых проблем с производительностью при одновременной отправке всех запросов);
- *сжатие HTTP-заголовков* (появилась возможность сжимать не только тела, но и заголовки HTTP-запросов).

С помощью текстового протокола запроса и ответа внедрить эти концепции не представлялось возможным, поэтому SPDY стал бинарным протоколом. Благодаря этому одиночное соединение получило возможность обрабатывать небольшие сообщения, которые вместе формировали более крупные HTTP-сообщения, во многом так же, как TCP сам разбивает большие HTTP-сообщения на множество меньших TCP-пакетов, которые прозрачны для большинства реализаций HTTP. SPDY реализовал концепции TCP на уровне HTTP, так что теперь благодаря ему несколько HTTP-сообщений могут передаваться одновременно.

Дополнительные расширенные функции, такие как отправка по инициативе сервера, позволяли серверу добавлять дополнительные ресурсы. Если вы запросили домашнюю страницу, в ответ push-сервер может предоставить файл CSS, необходимый для ее отображения. Этот процесс предотвращает задержки производительности при запросе CSS-файла (в обе стороны), а также упрощает встраивание критически важных CSS.

Компания Google в то время находилась в уникальном положении, так как контролировала как браузер (Chrome), так и некоторые из самых популярных веб-сайтов (например, www.google.com). Вследствие этого компания имела возможность проводить масштабные эксперименты с новым протоколом на реальных веб-сайтах на обеих сторонах соединения. SPDY был выпущен для Chrome в сентябре 2010 года, а уже к январю 2011 года все сервисы Google получили поддержку SPDY[1].

SPDY возымел почти мгновенный успех, и в скором времени его начали поддерживать другие браузеры и серверы. Сначала в 2012 году это сделали Firefox и Opera, затем сервер Jetty и другие серверы, включая Apache и nginx. Подавляющее большинство веб-сайтов, поддерживающих SPDY, находились на двух последних веб-серверах. Веб-сайты, поддерживающие SPDY (например, Twitter, Facebook и WordPress), показали такой же прирост производительности, что и Google, с минимальными неудобствами, если не считать необходимость начальной установки. По данным w3techs.com[2], SPDY охватил до 9,1 % всех веб-сайтов. Сегодня, когда появился HTTP/2, некоторые браузеры перестали поддерживать SPDY. С начала 2018 года использование SPDY резко упало (рис. 2.17).

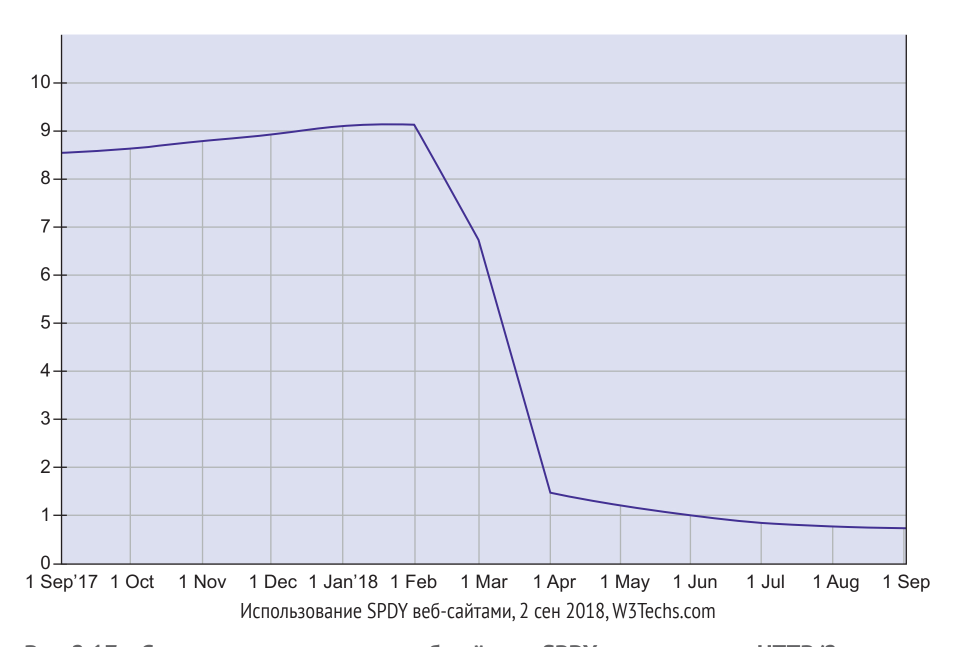


Рис. 2.17 Сокращение поддержки веб-сайтами SPDY после запуска HTTP/2

1 https://groups.google.com/d/msg/spdy-dev/TCOW7Lw2scQ/T2kM5aPDydwJ.

2 https://w3techs.com/technologies/details/ce-spdy/all/a.

2.5.2 HTTP/2

SPDY доказал возможность улучшения HTTP/1.1 не только теоретически, но и на реальных примерах. В 2012 году рабочая группа HTTP в составе IETF отметила успех SPDY и запросила предложения по следующей версии HTTP[1]. SPDY стал основой для следующей версии. Однако рабочая группа предпочитала оставаться открытой для любых предложений (хотя некоторые люди оспаривают эту позицию, как описано в главе 10).

Какое-то время рассматривались и другие предложения, но в конечном итоге в ноябре 2012 года[2] был создан первый проект, где в основу HTTP/2 SPDY лег именно SPDY. Впоследствии проект претерпел некоторые изменения и улучшения (в частности, в плане использования потоков и сжатия). В главах 4, 5, 7 и 8 мы рассмотрим технические детали этой версии намного подробнее.

К концу 2014 года спецификация HTTP/2 была представлена в качестве стандарта, а в мае 2015 года она была официально утверждена как RFC 7450[3]. Так как спецификация основывалась на SPDY, многие серверы почти сразу же ввели поддержку новой версии. С февраля 2015 года поддержку HTTP/2 ввел Firefox, а с марта 2015-го Chrome и Opera. Чуть позже, в этом же году, к ним примкнули Internet Explorer 11, Edge и Safari.

Веб-серверы быстро адаптировались к новому протоколу, и многие из них реализовали различные версии по мере их стандартизации. Одними из первых реализациями были LiteSpeed[4] и H2O[5]. К концу 2015 года три основных веб-сервера, используемых подавляющим большинством Internet-пользователей (Apache, IIS и nginx), создали свои реализации, хотя изначально они были отмечены как экспериментальные.

По данным w3tech.com[6], по состоянию на сентябрь 2018 года HTTP/2 был доступен на 30,1 % всех веб-сайтов. Такой охват во многом связан с сетями доставки контента и более крупными сайтами, поддерживающими HTTP/2. Для столь молодой технологии это блестящий результат. Как вы увидите в главе 3, использование HTTP/2 на стороне сервера в настоящее время требует изрядных усилий.

Сегодня протокол HTTP/2 уже доступен и работает на практике. Вскоре стало очевидно, что данный протокол значительно улучшает производительность именно за счет того, что решает проблемы с HTTP/1.1, которые мы обсуждали в этой главе.

---

1 https://lists.w3.org/Archives/Public/ietf-http-wg/2012JanMar/0098.html.

2 https://tools.ietf.org/html/draft-ietf-httpbis-http2-00.

3 https://tools.ietf.org/html/rfc7540.

4 https://blog.litespeedtech.com/2015/04/17/lsws-5-0-is-out-support-for-http2-esi-litemage-cache/.

5 https://h2o.examp1e.net/.

6 https://w3techs.com/technologies/details/ce-http2/all/all.

2.6 Значение HTTP/2 для веб-производительности

Мы познакомились с проблемами производительности, возникающими при использовании HTTP/1, и узнали, что они могут быть решены с помощью HTTP/2, но все ли проблемы с Internet-производительностью решаются с помощью HTTP/2, и на какой прирост скорости мы можем рассчитывать?

2.6.1 Пример предельной производительности HTTP/2

Существует множество примеров, демонстрирующих улучшение производительности при использовании HTTP/2. Одним из них является мой персональный сайт https://www.tunetheweb.com/performance-test-360/. Он доступен при использовании HTTP 1.1, HTTP 1.1 через HTTPS и HTTP/2 через HTTPS. Как было сказано в главе 3, браузеры поддерживают HTTP/2 только при HTTPS-соединении, следовательно, тестирование HTTP/2 невозможно без использования протокола HTTPS. Мы можем провести анализ с помощью https://www.httpvshttps.com/. Данный сайт загружает 360 уникальных изображений по трем технологиях (HTTP, HTTPS, HTTP/2) с использованием JavaScript. Затем можно сравнить результаты. Результат проведенного нами теста показан на рис. 2.18.

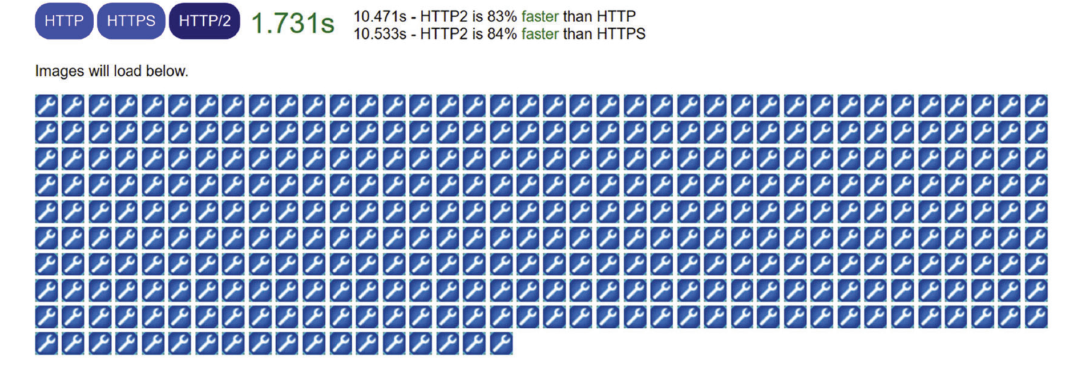


Рис. 2.18 Сравнение HTTP, HTTPS и HTTP/2

Согласно проведенному тесту, версии HTTP для загрузки страницы и всех изображений понадобилось 10,471 с, а версии HTTPS – 10,533 с. Исходя из этого, мы видим, что использование HTTPS практически не снижает производительность, а разница между ним и текстовым протоколом практически не видна. Данный тест показал, что HTTPS работает ненамного быстрее, чем HTTP, что фактически не имеет смысла (поскольку HTTPS требует дополнительной обработки). Для данного сайта дополнительная обработка находится в пределах погрешности.

HTTP/2 загрузил сайт за 1,731 с, что на 83 % быстрее, чем HTTP и HTTPS. На каскадных диаграммах видна причина этого. Сравните диаграммы HTTPS и HTTPS/2 на рис. 2.19 и 2.20.

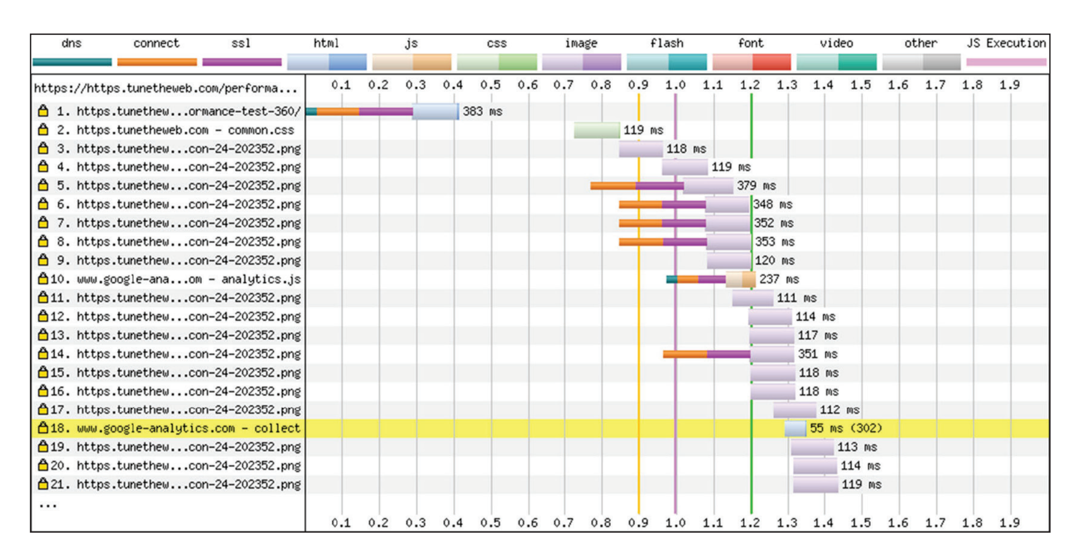


Рис. 2.19 **Каскадная диаграмма теста HTTPS (игнорируйте выделенную строку 18 с ответом 302)**

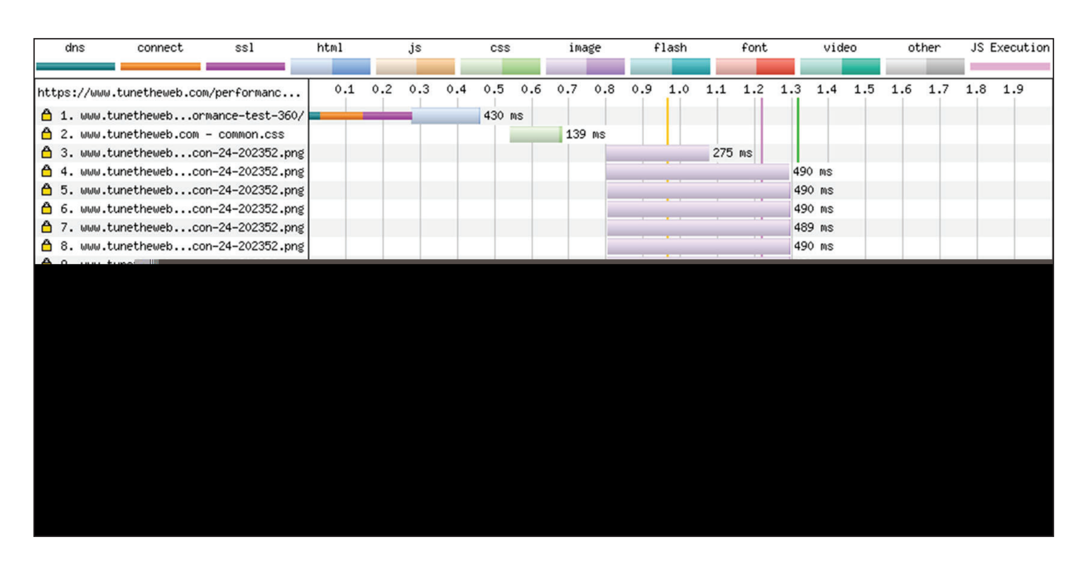


Рис. 2.20 **Каскадная диаграмма теста HTTP/2**

При использовании HTTPS происходит уже знакомая нам задержка, возникающая при установке дополнительных соединений и последующей загрузке изображений партиями по 6 штук. Однако при использовании HTTP/2 все изображения запрашиваются одновременно, и задержки не происходит. Для краткости примера мы решили показать только первый 21 запрос, а не все 360. Однако даже на кратком примере заметно существенное преимущество HTTP/2 при загрузке сайтов такого типа. Также обратите внимание на рис. 2.19. На нем показано, что в запросе 10, после того как было установлено максимальное количество со-

единений, браузер выбирает загрузку Google Analytics. Данный запрос относится уже к другому домену, где лимит возможных соединений еще не превышен. На рис. 2.20 мы видим, что параллельные запросы сильно ограничены, вследствие чего может быть запрошено гораздо больше изображений. Запрос Google Analytics в данной диаграмме находится ниже 21 запроса.

Внимательные читатели могут заметить, что загрузка изображений по протоколу HTTP/2 занимает больше времени (490 мс), по сравнению с HTTP/1.1 (115 мс), не учитывая загрузку первые 6 запросов. Время загрузки ресурсов по протоколу HTTP/2 может разниться из-за различных способов измерения. В каскадных диаграммах время загрузки обычно измеряется с момента ответа отправки запроса до момента получения ответа и не включает в себя время ожидания в очереди. К примеру, в запросе 16 при работе с HTTP/1 ресурс запрашивается примерно на отметке 1,2 с, а ответ приходит через 118 мс (1,318 мс). В данном случае изображение было для браузера необходимым ресурсом. Таким образом, браузер обработал HTML и совершил первый запрос уже через 0,75 с, что совпадает с цифрами, которые мы имеем при использовании HTTP/2. Следовательно, задержка 0,45 с в каскадных диаграммах HTTP/1 не отображается, и, возможно, отсчет должен начинаться с отметки 0,75. Как отмечалось в разделе 2.4.2, каскадная диаграмма Chrome включает в себя информацию о времени ожидания; таким образом, на ней мы можем наблюдать истинное общее время загрузки, которое дольше, чем у HTTP/2.

Однако при использовании HTTP/2 в некоторых случаях запросы *могут занимать больше времени* из-за низкой пропускной способности у клиента или сервера. Необходимость использовать несколько соединений в HTTP/1 всегда создает очереди из 6 запросов. HTTP/2 использует лишь одно соединение, разделенное на потоки. Теоретически в этом случае ограничения устраняются, однако во многих реализациях могут возникать иные ограничения. Например, страница из примера размещена на сервере Apache, который по умолчанию ограничивается 100 запросами на одно соединение. Отправка большого количества запросов одновременно приводит к тому, что запросы начинают использовать одни и те же доступные ресурсы, ввиду чего загрузка занимает больше времени. На рис. 2.20 мы видим, что время загрузки изображений постепенно увеличивается (если в строке запроса 4 оно составляет 282 мс, то в строке запроса 25 уже 301 мс). На рис. 2.21 показаны те же результаты в строках запросов 88–102. Здесь время запросов изображений составляет до 720 мс, что в 6 раз дольше, чем при использовании HTTP/1. Кроме того, по достижении предела в 100 запросов загрузка приостанавливается до тех пор, пока первые запросы не будут выполнены, и только после этого осуществляются остальные запросы. Происходит практически то же самое, что и при использовании HTTP/1. Однако при использовании HTTP/2 такой эффект возникает намного реже, а если и возникает, то несколько позже, так как лимит запросов здесь увеличен. Обратите внимание, что во время этой паузы в запросе 104 запрашивается Google Analytics. На рис. 2.19 мы видим аналогичный исход, но намного раньше, в запросе 10.

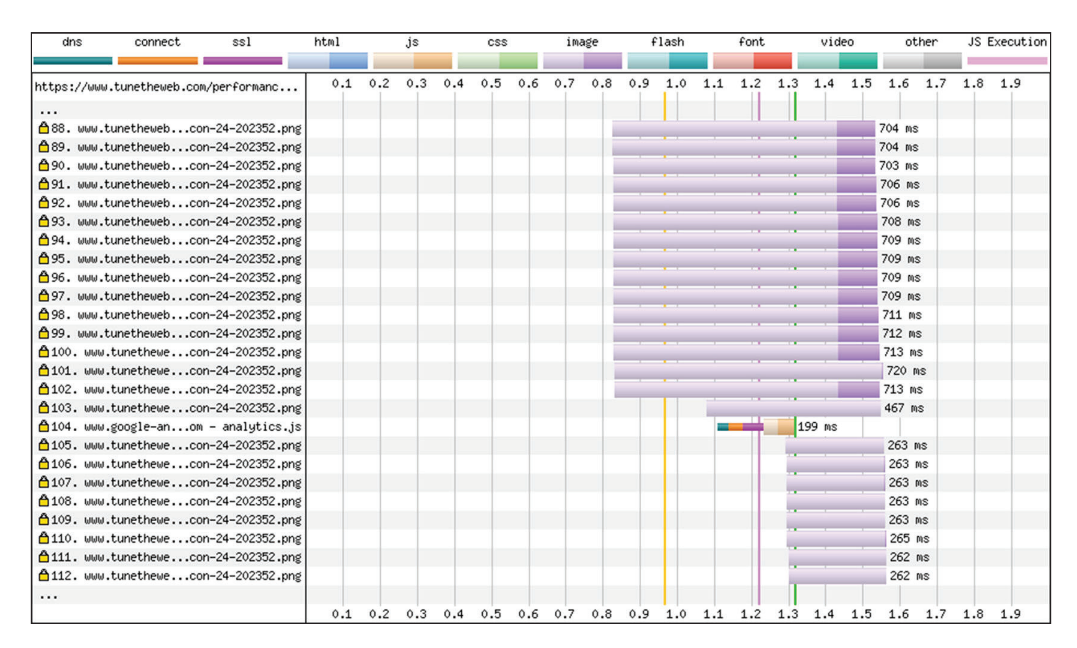


Рис. 2.21 Задержки в каскадной диаграмме HTTP/2

Глядя на диаграмму HTTP/2, можно упустить один важный момент, а именно то, что, в сущности, она является измерением иных аспектов. Однако для пользователя важно общее время загрузки, и здесь HTTP/2 выигрывает.

2.6.2 Какой прирост производительности может обеспечить HTTP/2?

Пример в разделе 2.6.1 показывает нам преимущества HTTP/2. Прирост производительности составляет 83 %, и это впечатляет. Однако такой исход возможен далеко не у всех веб-сайтов. В данном примере мы наблюдаем условия, при которых HTTP/2 дает наилучшие результаты (к слову, это еще одна причина, по которой лучше использовать реальные, широко используемые сайты). Некоторые сайты при переходе на HTTP/2 могут не получить такого прироста быстродействия в силу других недостатков, на фоне которых недостатки HTTP/1 не выглядят главной проблемой.

Существует две причины, по которым переход на HTTP/2 может не принести желаемых результатов. Первая причина состоит в том, что многие веб-сайты уже настроены на использование обходных путей, описанных в разделе 2.2, и поэтому недостатки HTTP/1 для них не столь заметны. Однако даже такие сайты подвержены некоторым проблемам с производительностью, так как обходные пути также имеют свои недостатки. Кроме того, для их внедрения требуются значительные усилия. В теории, HTTP/2 может обеспечить сайтам лучшую производительность путем эффективного использования спрайт-файлов и встроенных CSS, JavaScript и изображений, затрачивая при этом намного меньше усилий.

Вторая причина заключается в том, что некоторые веб-сайты имеют куда более серьезные проблемы с производительностью. Например, на многих из них размещено большое количество изображений в высоком качестве или слишком много JavaScript. Все это требует больших затрат времени для загрузки (здесь HTTP/2 может помочь) и обработки (здесь HTTP/2 бессилен). HTTP/2 не сможет помочь сайтам, которые работают медленно даже после загрузки или страдают от зависания (когда браузер изо всех сил пытается справиться с прокруткой страницы веб-сайта пользователем), так как данный протокол решает проблемы производительности только на сетевой стороне. Потеря пакетов (которую мы рассмотрим в главе 9) также может замедлить работу HTTP/2/, однако это крайние случаи.

Несмотря на все вышеупомянутое, мы уверены, что HTTP/2 крайне полезен для повышения производительности. Также его использование поможет избежать использования обходных путей. Но при этом важно понимать, какие проблемы HTTP/2 способен решить, а какие – нет. В противном случае пользователи могут заметить прирост производительности не сразу и разочароваться. На момент написания данной книги мы, вероятно, находимся на пике завышенных ожиданий (стоит упомянуть о циклах зрелости технологий от Gartner[1]). Сегодня пользователи ждут, что новая технология сможет решить все проблемы. Владельцам веб-сайтов следует тщательно изучить проблемы с производительностью их сайтов, ведь низкая пропускная способность в HTTP/1 – это лишь одно из узких мест. Впрочем, чаще всего после перехода на HTTP/2 производительность все же возрастает, так как HTTP/2 не должен быть медленнее, чем HTTP/1. Но иногда такие случаи все же происходят. Например, сайты с низкой пропускной способностью (на которых размещено множество высококачественных изображений) могут работать на HTTP/2 медленнее, если порядок запросов, обеспечиваемый ограниченным количеством соединений в HTTP/1.1, загружает критически важные ресурсы быстрее. Одна компания, специализирующаяся на графическом дизайне, опубликовала интересный пример[2]. Но даже в таких случаях работу сайта можно ускорить посредством правильной настройки HTTP/2 (см. главу 7).

Возвращаясь к реальным практическим примерам, взглянем на табл. 2.2. В ней приведено время загрузки копии сайта Amazon с помощью HTTP/1 и HTTP/2 (оба раза через HTTPS), где все ссылки были изменены на локальные.

Таблица 2.2 Прирост производительности сайта Amazon, полученный благодаря HTTP/2

Протокол	Время загрузки	Первый байт	Начать отрисовки	Визуальная готовность	Индекс скорости
HTTP/1	2,616	0,409 с	1,492 с	2,900 с	1692
HTTP/2	2,337	0,421 с	1,275 с	2,600 с	1317
Разница	11 %	–3 %	15 %	10 %	22 %

[1] https://www.gartner.com/technology/research/methodologies/hype-cycle.jsp.

[2] https://99designs.ie/tech-blog/blog/2016/07/14/real-world-http-2-400gb-of-images-per-day/.

В данной таблице представлены несколько терминов, широко используемых в области веб-производительности:

- *время загрузки* – это время, необходимое странице для отправки события `onload`, после загрузки всех CSS и блокирующих JavaScript;
- *первый байт* – это время, необходимое для получения первого ответа от веб-сайта. Обычно он представляет собой первый реальный ответ, не предусматривающий перенаправления;
- *начало рендеринга* – это момент начала визуализации страницы. Данный показатель является ключевым параметром производительности, так как, если у сайта возникнут проблемы с визуализацией, пользователи, скорее всего, откажутся от его использования;
- *завершение рендеринга* – это момент, когда страница перестает меняться. Зачастую асинхронный JavaScript все еще меняет вид страницы после события окончания загрузки;
- *индекс скорости* – это показатель WebPagetest, который указывает среднее время загрузки элементов страницы в мс[1].

HTTP/2 позволяет улучшить большую часть этих показателей. Время получения первого ответа немного увеличилось, но повторные тесты показали обратное, таким образом, можем сделать вывод, что этот результат находится в пределах погрешности.

Однако стоит признать, что данный пример создан искусственно, так как сайт реализован не в точности как у Amazon. Здесь используется только один домен (а не доменное разделение), а каждый ресурс сохранен как статический файл, а не динамический контент, как у Amazon (в таком случае возникли бы задержки). Тем не менее, несмотря на то что ограничения существуют и в HTTP/1, и в HTTP/2, в тестах HTTP/2 наблюдаются явные улучшения.

На рис. 2.22 и 2.23 мы видим каскадные диаграммы загрузок. В диаграмме HTTP/2 заметны ожидаемые улучшения: дополнительные подключения отсутствуют, а ступенчатая каскадная загрузка ресурсов в начале заметно меньше.

Код обоих типов запроса на веб-сайте не изменился благодаря HTTP/2. При загрузке сайта по данному протоколу мы все еще можем наблюдать эффект водопада, так как веб-технологии носят зависимый характер: например, для загрузки изображений веб страницам требуются CSS. Настройка соединений и время ожидания в очереди занимает меньше времени, поэтому эффект водопада, полученный в результате ограничений HTTP, исчезает. Цифры могут показаться небольшими, но повышение производительности на 22 % – это уже отличный результат, притом что фактически никаких серьезных изменений не потребовалось.

Сайты, оптимизированные для протокола HTTP/2 и успешно использующие его новые функции (о которых мы поговорим позже), в любом случае получат положительный результат. На данный момент у нас есть

---

[1] https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

20-летний опыт оптимизации сайтов под HTTP/1, но почти нет опыта оптимизации под HTTP/2.

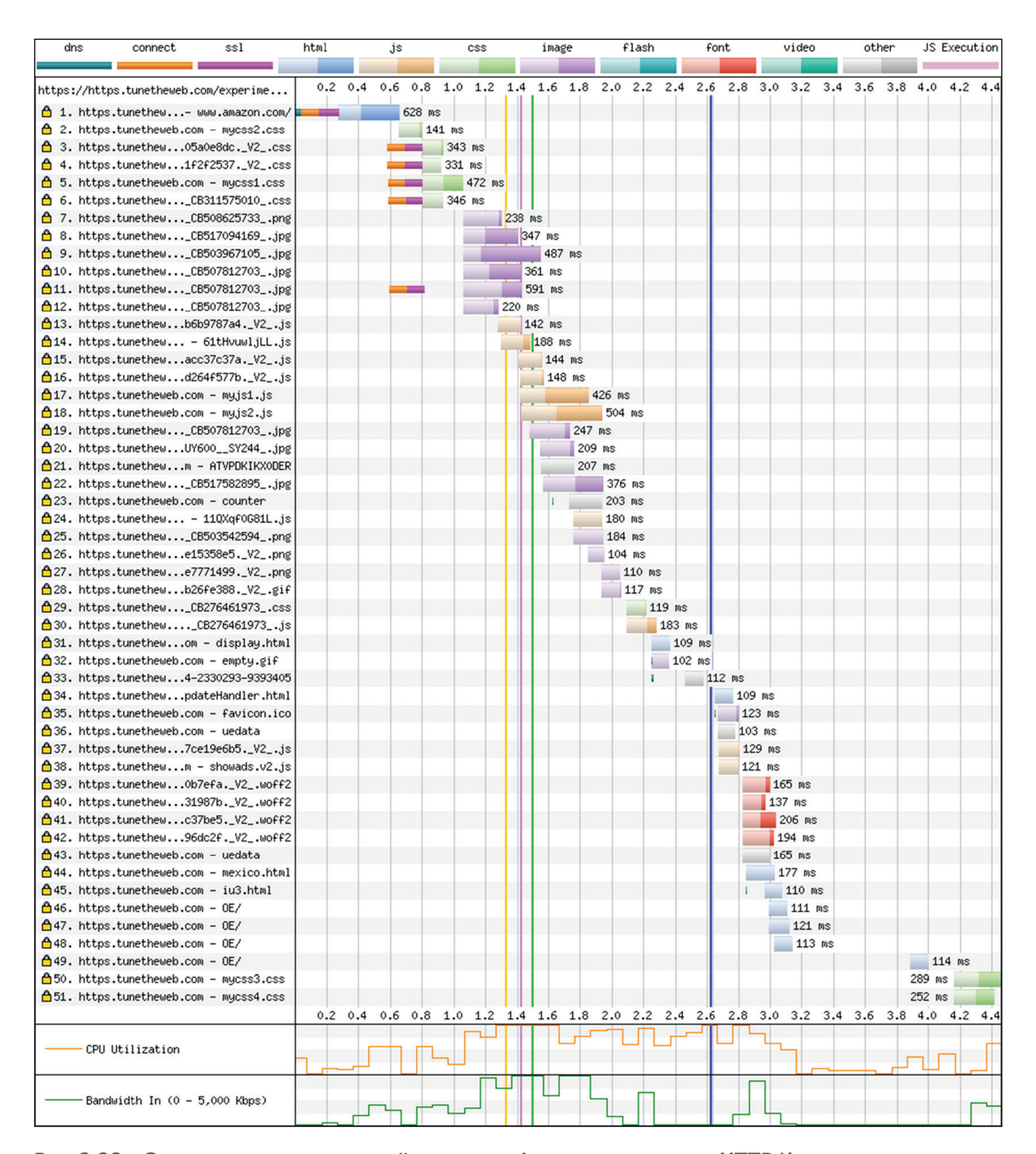


Рис. 2.22 Загрузка копии домашней страницы Amazon с помощью HTTP/1

В качестве примера мы используем Amazon, так как данный сайт очень популярен, но (на момент написания книги) еще не перешел на HTTP/2. Кроме того, его хорошо (но не идеально!) оптимизировали для HTTP/1. Мы не хотим сказать, что сайт Amazon написан или работает плохо. Мы лишь показываем улучшения, которые может внести HTTP/2, не прибе-

гая к обходным путям HTTP/1, на реализацию которых затрачиваются значительные усилия.

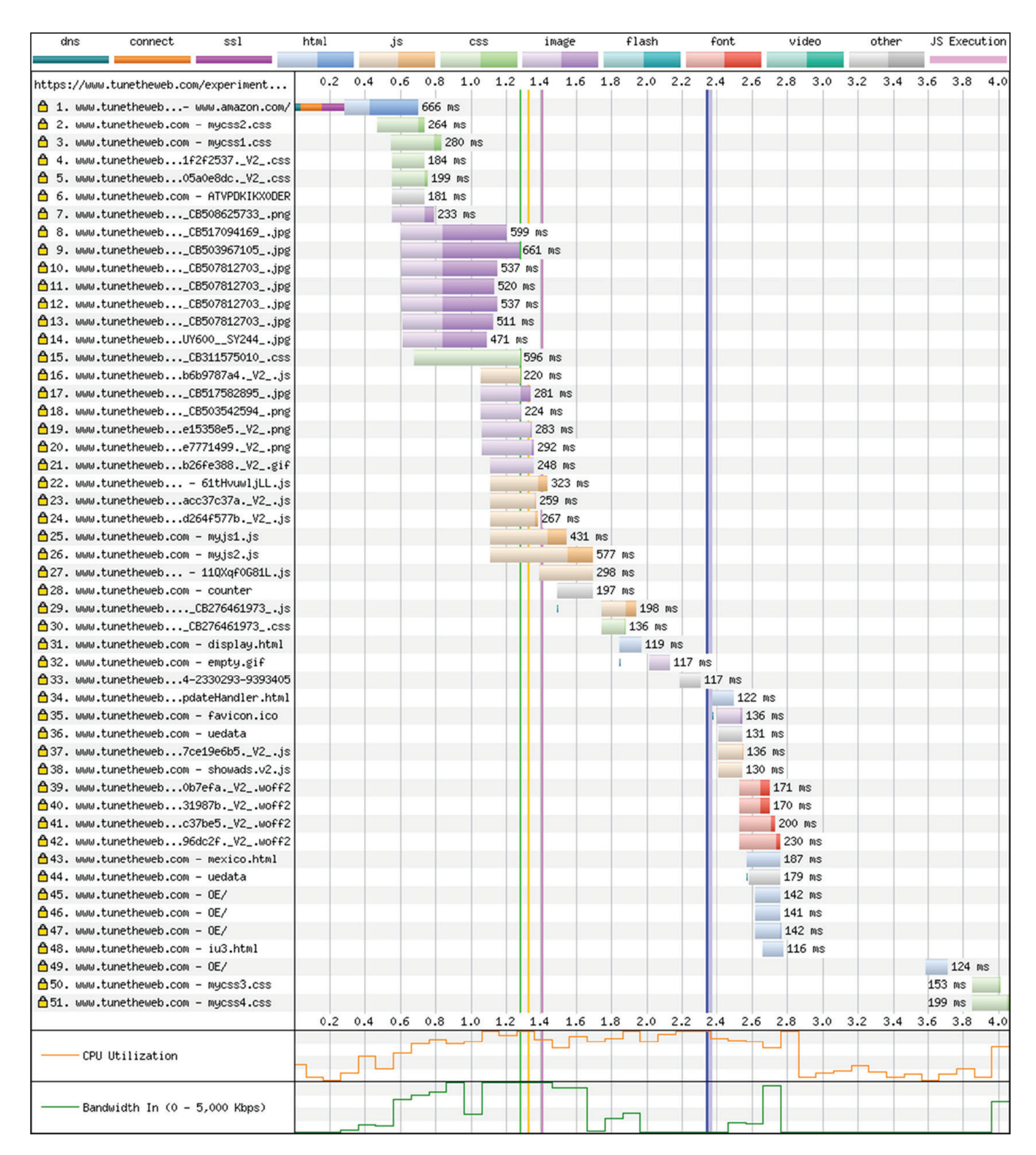


Рис. 2.23 Загрузка копии домашней страницы Amazon с помощью HTTP/2

С момента начала написания этой главы Amazon уже перешел на HTTP/2, и результаты перехода получились аналогичны результатам, полученным из наших примеров. Однако мы рассматривали его как реальный сложный веб сайт, который уже реализовал некоторые оптимизации производительности HTTP/1, но все еще не перешедший на HTTP/2.

2.6.3 Обходные пути для HTTP/1.1 как потенциальные тупики

В теории не должно быть необходимости раскрывать тему обходных путей HTTP/1, так как HTTP/2 устраняет проблемы предыдущей версии. Многие считают, что обходные пути становятся тупиками, так как могут помешать вам получить все преимущества HTTP/2. Преимущества использования одного TCP-соединения сводятся на нет, если владелец веб-сайта использует доменное разделение (в главе 6 мы поговорим об объединении подключений, которое позволяет решить данную проблему). HTTP/2 позволяет по умолчанию создать высокопроизводительный сайт.

К сожалению, в действительности все не так просто. Как будет сказано в последующих главах (особенно в главе 6), отказываться от обходных путей еще слишком рано, ведь HTTP/2 еще не получил должного распространения. На стороне клиента пользователи все еще применяют HTTP/1.1. Они также могут использовать старые браузеры или прокси-серверы, которые еще не поддерживают HTTP/2 (например, антивирусные сканеры или корпоративные прокси).

Кроме того, реализации на сторонах и клиента и сервера все еще подвержены изменениям в поиске условий, при которых протокол будет работать эффективнее. После запуска HTTP/1.1 область оптимизации веб-производительности росла и процветала, а разработчики обучались оптимизировать веб-сайты для работы с HTTP. Мы надеемся, что протоколу HTTP/2 потребуется меньше времени. Пока что разработчики не привыкли к новому протоколу, и многие передовые практики и методы требуют основательного изучения.

Сейчас мы с нетерпением ждем повсеместного внедрения HTTP/2. В главе 3 мы поговорим о том, как перейти на него. Чуть позже мы вернемся к теме оптимизации производительности и рассмотрим, как можно проследить изменения при переходе на HTTP/2 и как использовать его наиболее эффективно. Эта глава должна была объяснить читателям преимущества нового протокола, и мы надеемся, что нам это удалось.

Резюме

- В HTTP/1.1 существуют проблемы с производительностью, особенно если речь идет о загрузке нескольких ресурсов.
- Существуют обходные пути для решения этих проблем (использование параллельных подключений, шардинг, спрайтинг и т. д.). Однако они также имеют свои недостатки.
- С помощью таких инструментов, как WebPagetest, вы можете сгенерировать каскадные диаграммы, на которых можно легко отследить проблемы производительности вашего веб-сайта.
- Для решения этих проблем был разработан SPDY.
- HTTP/2 – это стандартизированная версия SPDY.
- Не все проблемы с производительностью можно решить с помощью HTTP/2.

3 Переход на HTTP/2

В этой главе мы рассмотрим:

- какие браузеры и серверы поддерживают HTTP/2;
- различные пути перехода вашего сайта на HTTP/2;
- обратные прокси-серверы и CDN, их влияние на HTTP/2;
- устранение ошибок, по причине которых HTTP/2 не используется повсеместно.

В первых двух главах мы получили представление о самом протоколе HTTP и о том, как он используется в наши дни. Также мы выяснили, почему версия HTTP/2 была столь необходима. В этой главе речь пойдет о том, как правильно перевести ваш сайт на HTTP/2 и извлечь из этого максимальную пользу.

3.1 Поддержка HTTP

Относительно недавно, в мае 2015 года, HTTP/2 был официально признан интернет-стандартом. Как в случае со всеми новыми технологиями, разработчикам необходимо принять решение, когда лучше всего начать внедрение. Если начать внедрение слишком рано, технология будет считаться передовой и рискованной, так как в будущем, она, вероятно, претерпит изменения или даже окажется неудачной. К тому же пользователи, использующие старые стандарты, могут быть не готовы поддерживать новую технологию. Это означает, что не стоит торопиться

с внедрением технологий в реальный мир. Но, с другой стороны, именно люди, впервые перешедшие на новую технологию, открывают для нее путь в массы.

К счастью, развитие HTTP/2 проходит не совсем по обычному технологическому циклу, ведь ранее эффективность протокола отчасти была доказана в его предыдущем воплощении SPDY (как было сказано в главе 2). По данным w3tech.com[1], на момент написания нашей книги уже более 30 % всех веб-сайтов перешли на использование нового стандарта. К тому времени, как вы прочитаете эту книгу, цифра, вероятно, увеличится. Эффективность HTTP/2 уже доказана. Многие сайты уже перешли на использование данного протокола.

Возможность использования новой-веб технологии на вашем сайте сводится к трем аспектам:

- поддерживается ли технология веб-браузерами?
- возможен ли переход на нее с вашими техническими возможностями?
- существует ли надежный запасной вариант на случай, если технология вам не подойдет?

В целом HTTP/2 соответствует всем трем аспектам. Данную технологию поддерживают практически все браузеры и серверные ПО. Кроме того, если ваш веб-сайт не сможет поддерживать HTTP/2, вы всегда сможете вернуться к HTTP/1.1. Однако существуют некоторые тонкости и нюансы.

3.1.1 Поддержка HTTP/2 со стороны веб-браузера

Практически все существующие в наши дни веб-браузеры поддерживают HTTP/2/. Мы можем сделать такой вывод, основываясь на данных caniuse.com[2] (рис. 3.1).

На западе позже всех на поддержку нового стандарта в своем веб-браузере перешла платформа Android. Однако на момент написания нашей книги переход на HTTP/2 все еще не осуществил браузер UC, популярный в Китае, Индии, Индонезии и других азиатских странах. Браузер Opera Mini получает готовый рендер страниц с серверов Opera, поэтому имеет мало отношения к нашему обсуждению.

На рис. 3.2 показано, как HTTP/2 используется веб-браузерами. Величина графы указывает на процент пользователей новой версии. Как мы можем заметить, браузер UC использует довольно большое количество пользователей. Ввиду того, что он не поддерживает HTTP/2, для распространения протокола это является серьезным препятствием.

---

[1] https://w3techs.com/technologies/details/ce-http2/all/all.

[2] https://caniuse.com/#feat=http2.

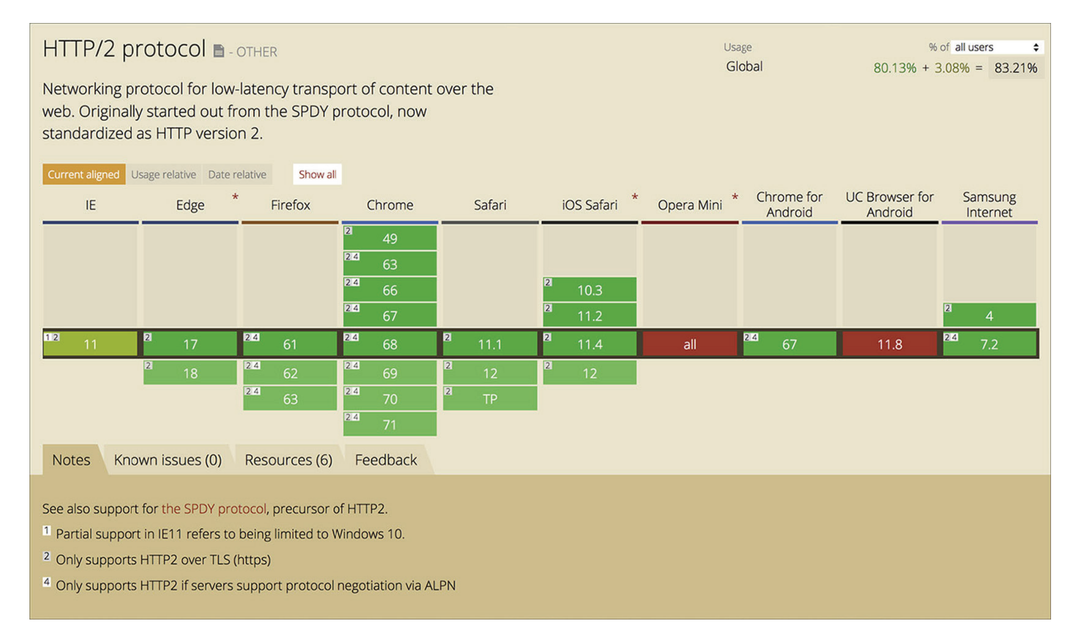


Рис. 3.1 Статистика caniuse.com для HTTP/2

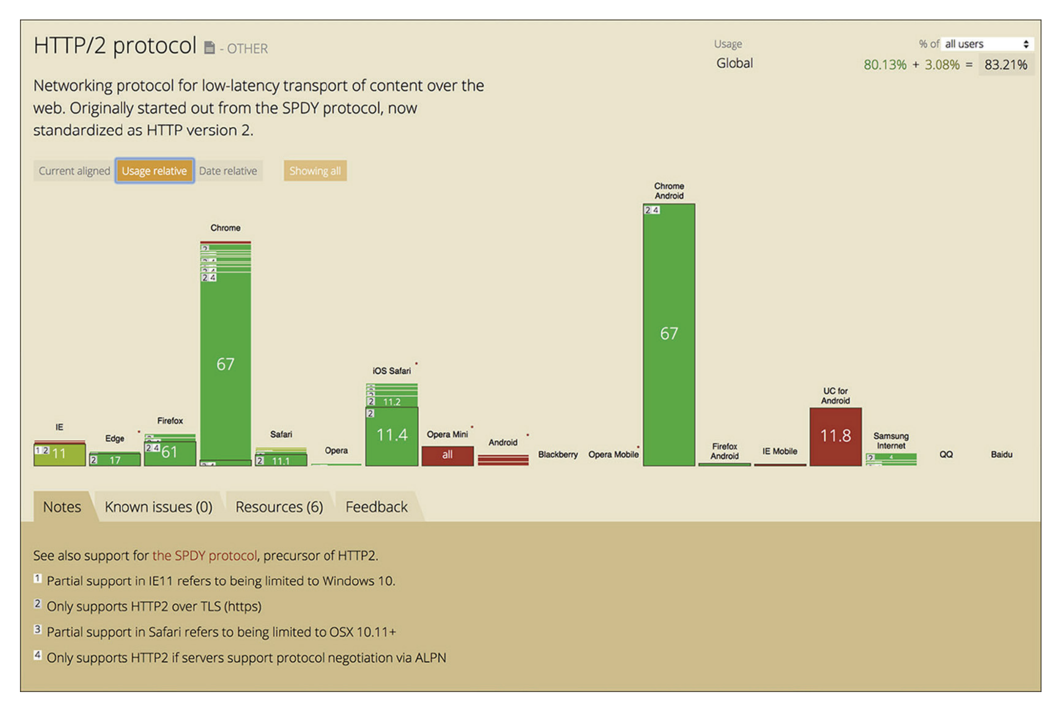


Рис. 3.2 Относительная поддержка HTTP/2 браузерами согласно caniuse.com

HTTP/2 поддерживают не все браузеры. Однако на момент написания книги около 83,21 % браузеров по всему миру уже перешли на использование данной версии, и со временем их количество будет только расти. Caniuse.com позволяет также просматривать статистику использования по странам. Таким образом, если вы обслуживаете пользователей какой-либо одной страны, вы можете получить еще более точную статистику по вашей пользовательской базе (исходя из которой может выясниться, что ваши пользователи используют UC не так уж и часто). В этой статистике, однако, существует несколько важных нюансов.

HTTP/2 и HTTPS для браузеров

На приведенных выше рисунках маленькой цифрой «2» отмечены браузеры, которые поддерживают HTTP/2 только через TLS (https). Таким образом, веб-сайты, использующие незащищенные подключения, не могут получить все преимущества от использования HTTP/2. Такая же ситуация была и со SPDY, поэтому многие настаивали на том, чтобы HTTPS был включен в официальную спецификацию HTTP/2. В конечном итоге это требование не включили в официальный документ, однако все создатели браузеров заявили, что они будут поддерживать HTTP/2 только вместе с HTTPS, что сделало это требование стандартом де-факто. Конечно же, такой стандарт, несомненно, огорчит владельцев веб-сайтов, использующих HTTP. Однако в пользу использования HTTPS есть два веских аргумента.

Первый из них носит исключительно практический характер. Использование HTTP/2 через HTTPS снижает вероятность возникновения проблем с совместимостью. Многие сервисы в Internet, поддерживающие HTTP, не смогут работать с сообщениями HTTP/2. При отправке через HTTPS-соединение HTTP-сообщения скрываются, тем самым предотвращаются проблемы совместимости. (HTTPS-сообщение может быть прочитано только получателем, но в следующем разделе мы рассмотрим особые случаи перехвата прокси.)

Второй аспект носит идеологический характер. Многие создатели браузеров (и другие пользователи, в том числе автор данной книги) верят в то, что отказ от незашифрованного HTTP-соединения возможен, и считают, что все веб-сайты должны поддерживать исключительно HTTPS. Поэтому внедрение новых функций зачастую требует использования HTTPS-соединения.

HTTPS обеспечивает безопасность, конфиденциальность и целостность соединения с веб-сайтами. Данные аспекты важны для всех сайтов, а не только для электронных торговых площадок, где необходимо защищать платежные данные[1]. История поиска и просматриваемых страниц является конфиденциальными личными данными. Различные регистрационные формы запрашивают ваш адрес электронной почты, т е. личную информацию, поэтому такие формы должны быть защищены. Для формата блога перехват и изменение данных кажется маловероятным.

[1] https://tools.ietf.org/html/rfc7258.

Однако операторы мобильной связи и Wi-Fi в самолетах размещают на страницах рекламные объявления, которые видят их пользователи при использовании незащищенного соединения. Многие злоумышленники могут распространять более опасный контент, например код JavaScript или вредоносное ПО для майнинга криптовалюты.

Со временем владельцам веб-сайтов избегать использования HTTPS будет все сложнее. HTTP/2, требующий защищенного соединения, является еще одной причиной для перехода.

Однако даже при использовании HTTPS у вас могут возникнуть некоторые проблемы. HTTP/2 нуждается в надежном HTTPS. Маленькая цифра «4» на рисунках выше отмечает браузеры, поддерживающие HTTP/2 только в случае поддержки серверами согласования протокола через ALPN. Такими браузерами являются Chrome, Firefox и Opera. В разделе 3.1.2 мы обсудим эту тему подробнее, но пока имейте ввиду, что *согласование протокола уровня приложения* (application layer protocol negotiation, ALPN) поддерживается только серверами, использующими HTTPS, и что некоторые браузеры не смогут использовать HTTP/2, если ALPN для них недоступен. Кроме того, многие браузеры требуют новых и безопасных наборов шифров для использования HTTP/2[1].

Перехват прокси

Для использования HTTP/2 требуется поддержка этой версии и браузером, и сервером. Однако, если прокси пользователей разрывают HTTP-соединение и к тому же не поддерживают HTTP/2, может возникнуть серьезная проблема.

Во многих корпоративных средах распространено использование прокси и ограничение прямого доступа в Internet. Такой подход позволяет сканировать соединения на наличие угроз и предотвращать доступ сторонних лиц к определенным сайтам (например, к личным учетным записях электронной почты). Существуют прокси-серверы и для домашнего пользования. Их создают антивирусные продукты, через которые проходит и сканируется на наличие угроз веб-трафик.

Для HTTPS-трафика такие прокси представляют серьезную проблему, поскольку они не могут читать зашифрованные сообщения. Таким образом, если вы используете прокси-сервер, которому необходимо читать HTTPS-трафик, ваш браузер будет создавать соединение с прокси, а прокси в свою очередь создаст отдельное HTTPS-соединение с веб-сайтом. Получается, что прокси отправляет поддельный сертификат HTTPS, выдавая себя за настоящий сайт. Обычно браузеры предупреждают об этом, поскольку суть HTTPS отчасти заключается в проверке подлинности издателя сертификата. Установка таких прокси включает в себя настройку программного обеспечения прокси в качестве утвержденного поставщика сертификатов на компьютере, для того чтобы веб-браузер принимал их поддельные сертификаты.

---

1 https://tools.ietf.org/html/rfc7540#appendix-A.

Разделение трафика на две части позволяет прокси читать его, но ваш браузер фактически не подключается напрямую к веб-сайту. В таком случае вы сможете использовать HTTP/2 только тогда, когда его поддерживает прокси. Если прокси не поддерживает HTTP/2, получается, что он переводит ваше соединение на HTTP/1.1. Это приводит к тому, что, во-первых, вы уже не используете HTTP/2, а во-вторых, возникает путаница. Вы не понимаете почему произошел переход на более раннюю версию, если и браузер, и сервер поддерживают HTTP/2 (см. раздел 3.3).

Многие специалисты, занимающиеся Internet-безопасностью, считают, что посредничество прокси вызывает больше проблем, чем решает, так как создатели браузеров активно работают над улучшением использования HTTPS, а разрыв соединения означает, что все это делается впустую. Тем не менее прокси все еще используются, и это необходимо учитывать при попытках разобраться в переходе на HTTP/2. Исследования показали, что из-за использования прокси перехватывается от 4 до 9 % всего трафика, причем 58 % этого трафика перехватывается антивирусным ПО, а 35 % – корпоративными прокси[1]. Самый простой способ узнать, использует ли компьютер прокси, – посмотреть сертификат HTTPS и удостовериться в его подлинности, является ли он продуктом настоящего центра сертификации или частью ПО. На рис. 3.3 показано различие между прямым подключением и перехватом подключения антивирусным сканером Avast в Internet Explorer.



Рис. 3.3 Сертификат HTTPS для прямого подключения к Google и подключения, перехватываемого антивирусным продуктом

Если взглянуть более позитивно, перехватывающие прокси обычно используются в домашних или корпоративных средах, где соединение в любом случае довольно быстрое, а использование HTTP/2 менее эффективно. Перехват мобильного трафика встречается реже, поэтому сети с малой задержкой (например, мобильные) больше других нуждаются в переходе на HTTP/2.

Вывод о поддержке HTTP/2 браузерами

Многие браузеры поддерживают HTTP/2. Появление так называемых вечнозеленых браузеров, которые обновляются для использования новых версий интернет-протоколов автоматически (см. сноску ниже), свидетельствует о том, что развертывание HTTP/2 на стороне браузеров

[1] https://jhalderm.com/pub/papers/interception-ndss17.pdf.

произошло довольно легко. Однако для эффективного использования HTTP/2 существует несколько условий. Одними из них являются правильная настройка HTTPS на стороне сервера, а также в случае использования перехватывающих прокси-серверов.

Факт того, что для использования HTTP/2 требуется HTTPS, а также строгий характер данного протокола, сильно усложняет переход на новую версию и может запутать пользователя. Данная сложность присуща только стороне сервера, как уже было сказано в 3.1.2. Постепенно все серверы переходят на HTTPS. Недостатки использования незашифрованных HTTP-соединений со временем будут становится все более явными. На момент написания данной книги более 75 % интернет-трафика осуществляется по HTTPS-соединениям[1]. Однако эта цифра несколько неточна ввиду высокой посещаемости некоторых крупных веб-сайтов. В любом случае, если ваш веб-сайт все еще не перевели ваш сайт на HTTPS, то в скором времени вам необходимо будет сделать это.

«Вечнозеленые» браузеры

Такие браузеры, как Chrome и Firefox, обновляются автоматически в фоновом режиме без запроса пользователя. Такие браузеры называются *«вечнозелеными»*. Пользователи, использующие последние версии этих браузеров, вероятно, будут использовать и HTTP/2.

Такие браузеры существовали не всегда. В истории веб-разработки было много случаев, когда разработчики были вынуждены определять версии браузеров и внедрять различные хаки для пользователей, которые, к примеру, все еще использовали Internet Explorer 5 или подобный браузер.

К сожалению, на деле все не так радужно. Chrome, Firefox и Opera неплохо работают на ПК и относятся к категории «вечнозеленых», однако другие браузеры и платформы не обновляются автоматически. Обновления Safari зачастую происходят вместе с обновлением операционной системы, в том числе и на мобильных устройствах. Операционная система iOS обновляется довольно часто, но глобальные обновления происходят один раз в год. Такие технологии, как HTTP/2, обычно входят в глобальные обновления. Android перешел на «вечнозеленый» Webview Chromium в версии Android 5 (Lollipop), однако пользователям все же часто приходится устанавливать обновления через Play Store. Еще один известный «вечнозеленый» браузер – это Edge. Сейчас его обновления связаны с обновлениями операционной системы[a], но Microsoft пообещали в скором времени исправить это[b].

Случается, так, что пользователи сами отключают функцию автоматического обновления. Многие корпоративные сети отказываются от автоматических обновлений, но позже не уделяют должного внимания развертыванию обновлений вручную.

---

[a] https://www.scirra.com/blog/173/just-how-evergreen-is-microsoft-edge.

[b] https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-theweb-better-through-more-open-source-collaboration/.

---

[1] https://letsencrypt.org/stats/.

3.1.2 Поддержка HTTP/2 серверами

Поддержка HTTP/2 серверами развивалась не так быстро, как браузерами, но сегодня практически все серверы добавили поддержку новой версии. На сайте Github мы можем найти список реализаций HTTP/2 как на стороне клиента, так и на стороне сервера[1]. Исходя из этого, мы легко можем отследить, какие серверы поддерживают новый протокол. По данным Netcraft[2], наиболее популярные веб-серверы (на которые приходится более 80 % активных сайтов в Internet) – это Apache, nginx, Google и Microsoft IIS. Каждый из этих серверов поддерживает версию HTTP/2.

Успешное применение HTTP/2 на стороне сервера зависит не от того, поддерживает ли протокол последняя версия его программного обеспечения, а от того, поддерживает ли ее та версия, на которой работают веб-сайты в данный момент. Большинство реализаций не имеют функции автоматического обновления, и сделать ее не так легко, как для браузеров. В связи с этим на серверах зачастую используются более старые версии программного обеспечения, созданные задолго до выхода HTTP/2. Версии программного обеспечения очень часто привязаны к операционной системе (например, поддержка Microsoft IIS была добавлена только в IIS 10.0 в Windows Server 2016) или к диспетчеру пакетов операционной системы (например, диспетчер yum для Red Hat/CentOS/Fedora, где на момент написания нашей книги не установлена версия Apache или ngix с поддержкой HTTP/2).

Процесс обновления версии серверного программного обеспечения может быть довольно сложным. В системах на Linux для обновления требуется загрузка исходного кода и его компиляция, что требует соответствующих навыков и квалификации разработчиков, которые, кроме того, должны поддерживать актуальность ПО или как минимум следить за обновлениями патчей безопасности. В том, что процессом обновления руководит операционная система или диспетчер, кроется большое преимущество, состоящее в том, что проблемы безопасности разрешаются с каждым новым обновлением, выходящим периодически (или даже автоматически). Решение проблем безопасности вне обновлений системы требует соответствующих затрат и проведения серьезной работы, и к тому же существует большое количество рисков, связанных с этим. При работе с некоторыми ОС возможно использовать сторонние репозитории, предоставляющие новые версии ПО. Однако в таком случае речь идет об отказе от официальных репозиториев в пользу третьей стороны.

Криптографические библиотеки и их поддержка

Одна из самых больших проблем, возникающих на стороне сервера, – строгие требования, предписывающие использование HTTPS (особенно это характерно для Linux), о чем мы говорили ранее в этой главе. Большинство серверов используют для работы с тонкостями HTTPS отдельные

1 https://github.com/http2/http2-spec/wiki/Implementations.
2 https://news.netcraft.com/archives/category/web-server-survey/.

библиотеки – обычно это OpenSSL, хотя существуют и другое варианты, такие как LibreSSL и BoringSSL. Зачастую криптографическая библиотека является частью операционной системы. Обновление сервера – сложный процесс, а обновление SSL/TLS представляет собой еще более серьезную задачу, так как оно потенциально может повлиять на остальное программное обеспечение сервера.

В разделе 3.1.1 было сказано, что Chrome и Opera поддерживают HTTP/2 только с помощью расширения для SSL/TLS ALPN, а более раннее расширение NPN (Next Protocol Negotiation) для них не подходит. ALPN, как и NPN, позволяет серверу указывать, какие протоколы приложений сервер поддерживает в рамках согласования HTTPS; более подробно мы поговорим об этой теме в главе 4. Проблема заключается в том, что поддержка ALPN доступна только в последних версиях OpenSSL (версия 1.0.2 и новее), ввиду чего расширение недоступно на многих платформах. RedHat и CentOS добавили поддержку OpenSSL 1.0.2 только в августе и сентябре 2017 года соответственно. Однако коробочные версии программного обеспечения таких веб-серверов, как Apache, зачастую скомпилированы с использованием более старой версии 1.0.1, которая не позволяет использовать ALPN и, следовательно, HTTP/2 для Chrome и Opera. Точно так же Ubuntu добавил поддержку в версии 16, а не в версии 14, которая используется намного чаще. Debain начал поддерживать OpenSSL с ALPN в версии 9 (Stretch). Иногда, даже если у сервера существует библиотека OpenSSL, код HTTP/2 может быть не включен в нее по умолчанию. Обратите внимание на табл. 3.1.

Таблица 3.1 Поддержка ALPN операционными системами Linux

Операционная система	ALPN в OpenSSL по умолчанию	ALPN в Apache/nginx по умолчанию	HTTP/2 в Apache/nginx по умолчанию
RHEL/CentOS < 7.4	N (1.0.1)	N	N
RHEL/CentOS 7.4 & 7.5	Y (1.0.2)	N/Y	N/Y
Ubuntu 14.04 LTS	N (1.0.1)	N	N
Ubuntu 16.04 LTS	Y (1.0.2)	Y	N/Y
Ubuntu 18.04 LTS	Y (1.1.0)	Y	Y
Debian 7 (“Wheezy”)	N (1.0.1)	N	N
Debian 8 (“Jessie”)	N (1.0.1)	N	N
Debian 9 (“Stretch”)	Y (1.1.0)	Y	N
			Y

В таблице выше мы видим, что из всех представленных дистрибутивов Linux только коробочные версии Ubuntu 18.04 и Debian 9 содержат HTTP/2 для Apache (хотя его необходимо активировать во время настройки веб-сервера). Для RHEL/CentOS HTTP/2 для Apache необходимо устанавливать из какого-либо источника или из другого репозитория не по умолчанию.

Для nginx HTTP/2 может быть установлен через репозиторий nginx 9[1]. Таким образом, HTTP/2 обычно настраивается для nginx последних версий, однако настройка по-прежнему зависит от версии OpenSSL.

1 http://nginx.org/en/download.html.

Вывод о поддержке HTTP/2 серверами

В теории поддержка HTTP/2 серверами важна в той же степени, что и браузерами. Однако на практике большинство людей использует старые версии программного обеспечения серверов, которые не поддерживают HTTP/2. Эти версии требуют обновлений, которые в свою очередь могут быть простыми или сложными. Ситуация изменится, когда новые версии операционных систем будут использоваться повсеместно, однако это может стать проблемой для освоения HTTP/2. Хорошая новость в том, что инициаторами обновления должны стать владельцы веб-сайтов, которые могут принять на себя обязанности по обновлению своего программного обеспечения до версий, которые будут поддерживать HTTP/2, на стороне серверов. Как только эта работа будет проделана, большинство клиентского программного обеспечения также поддержит эти изменения. Если бы серверы поддерживали HTTP/2, а клиенты нет, владельцам сайтов так или иначе пришлось бы ждать, пока их пользователи обновят свое ПО. Если у вас нет возможности или желания менять свое программное обеспечение, то вы можете воспользоваться другими реализациями, которые мы обсудим в разделе 3.2.

3.1.3 Откат к предыдущим версиям, в случае если поддержка HTTP/2 невозможна

Если HTTP/2 не поддерживается вашим сервером, это никак не отразится на работе веб-сайтов, так как в таком случае они могут снова перейти на HTTP/1.1. Данная версия протокола будет продолжать использоваться еще долгое время. В теории если ваш сервер поддерживает HTTP/2, то его работа будет максимально эффективной.

Однако переход на HTTP/2 и сопутствующие ему изменения на веб-сайтах могут смутить пользователей HTTP/1.1. Сайт, конечно же, будет работать, но, возможно, медленнее, чем до этого. Насколько серьезна эта проблема для вас, зависит от того, какой у вас трафик HTTP/1.1. Мы вернемся к этой теме в главе 6.

Намного сложнее оценить проблемы реализации на стороне клиента или сервера. HTTP/2 все еще является достаточно молодой технологией и, несмотря на то что многие уже активно используют его на практике, пока что он находится на ранней стадии внедрения. Несомненно, в реализациях будут обнаружены ошибки, которые могут повлиять на загрузку вашего веб-сайта через HTTP/2. Такие ошибки обычно приводят к замедлению работы HTTP/2, однако серьезных проблем за собой не несут. Тем не менее, прежде чем переходить на HTTP/2 (или при установке любого другого крупного обновления), вам следует провести большое количество тестов.

3.2 Способы перехода вашего сайта на HTTP/2

Самое очевидное решение – просто подключить HTTP/2 на вашем веб-сервере, однако этот процесс может требовать обновления ПО. Существует еще несколько способов перехода. Например, вы можете просто добавить на свой сервер новый элемент инфраструктуры, например программу или сервис CND, который будет обрабатывать HTTP/2-соединения. Какой метод подойдет именно вам, зависит от нескольких факторов: от того, поддерживает ли ваш веб-сервер HTTP/2, насколько сложно для вас будет реализовать поддержку HTTP/2 и хотите ли вы усложнить среду реализацией некоторых других опций.

После подключения HTTP/2 вы можете увидеть, что трафик все еще передается с помощью HTTP/1.1. В разделе 3.3 мы обсудим устранение подобных неполадок. Если вы уже перевели свой сервер на HTTP/2, но он не работает в вашей среде, можете сразу перейти к данному разделу.

3.2.1 HTTP/2 на вашем веб-сервере

Подключение вашего сервера к HTTP/2 позволяет использовать новый протокол клиентам, которые имеют возможность поддерживать его. На рис. 3.4 показана простая схема подключения HTTP/2.

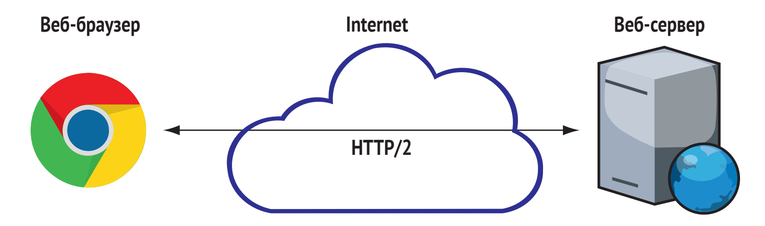


Рис. 3.4 HTTP/2 на вашем веб-сервере

Главная проблема состоит в том, что использование данного протокола на вашем сервере может быть недоступно. Как было сказано в разделе 3.1.2, вам может потребоваться обновить веб-сервер до новой версии, что повлечет за собой и обновление операционной системы, на которой работает ваш веб-сервер. Возможно и такое, что программное обеспечение вашего сервера не будет поддерживать HTTP/2 даже после обновления до новейшей версии. В табл. 3.2 приведен список некоторых широко известных веб-серверов и серверов-приложений и их версий, в которые была добавлена поддержка HTTP/2.

Таблица 3.2. Версии популярных серверов, в которые была добавлена поддержка HTTP/2

Веб-сервер	Добавленная версия HTTP/2
Apache HTTPD	2.4.17 (хотя отмечен как экспериментальный до 2.4.26)
IIS	10.0
Jetty	9.3
Netty	4.1
Nginx	1.9.5
NodeJS	8.4.0 (хотя не включен по умолчанию до 9.0 и отмечен как экспериментальный до 10.10)
Tomcat	8.5

Программное обеспечение Linux обычно устанавливается с помощью диспетчеров пакетов (таких как yum и aptget) при использовании официальных репозиториев, что упрощает установку и обновление программ. Многие из представленных в таблице серверов новым функциям предпочитают стабильность. Поскольку HTTP/2 является относительно молодым протоколом, версии веб-серверов по умолчанию обычно не включают в себя его поддержку. Если ваша операционная система не позволяет легко перевести сервер на HTTP/2, но вы хотите использовать именно эту версию протокола, у вас остается не так уж много вариантов, а установка приложений из неофициальных источников – это большой риск, и вы должны понимать возможные последствия, прежде чем идти по этому пути (см. сноску ниже).

Риски, возникающие при установке приложений из неофициальных источников

Установка приложений из неофициальных источников представляет собой загрузку готового пакета с другого сайта, добавление репозитория для диспетчера пакетов или установку исходного кода.

Загрузка готового пакета от третьего лица означает, что вы полностью доверяете ему ключевую часть своей инфраструктуры и root (как обычно это делают веб-серверы). Кроме того, многие готовые пакеты статически компилируются в соответствии с версией OpenSSL, поэтому, если в OpenSSL обнаружена уязвимость, вам необходимо обновить свой веб-сервер и исправить ошибки. Многих компаний не устраивает ни одно из этих ограничений. Если вам удобно использовать готовый пакет стороннего производителя, вы можете воспользоваться сайтом CodeIt[a], который предоставляет репозиторий с готовыми пакетами Apache и nginx и предоставляет инструкции по их установке.

Также вы можете «упаковать» свой сервер в контейнер, например с помощью Docker. Доступны образы контейнеров с соответствующими версиями обычных веб-серверов. В этом случае вы также доверяете свой сервер стороннему лицу, однако ввиду того, что вы «упаковываете» приложения в отдельный контейнер, для стороннего лица доступна лишь часть информации. Такой тип программного обеспечения является отдельной темой, поэтому мы не будем обсуждать его в нашей книге.

Для тех, кто не привык доверять третьим лицам, существует еще один вариант. Например, вы можете установить программное обеспечение из оригинального исходного кода. Исходный код должен принадлежать авторитетному источнику (в идеале создателю операционной системы). После загрузки код необходимо подтвердить либо путем проверки подписи, либо путем проверки хеша загрузки.

Даже если вы пользуетесь официальными источниками, вы управляете программным обеспечением уже не с помощью инструментов диспетчера пакетов, поэтому вам будут недоступны патчи безопасности, которые обычно устанавливает диспетчер; таким образом, вам придется делать это вручную. В стандартных репозиториях RHEL 7 и CentOS уже есть Apache 2.4.6. Однако это не оригинальная версия. Red Hat постоянно обновляет его, чтобы включить в новые версии все необходимые обновления безопасности. Установив версию вне диспетчера пакетов, вы лишитесь всех этих обновлений. Если вы не установите их вручную, вы рискуете получить небезопасное программное обеспечение, уязвимое для атак.

Еще один вариант – использование полуофициальных репозиториев. Многие создатели операционных систем предоставляют альтернативные репозитории для хранения программного обеспечения (например, Red Hat Software Collections). Кроме того, они могут предоставлять и официальные репозитории (как nginx). Здесь преимуществами являются простота установки и простота исправлений ошибок.

Таким образом, необходимо решить, какой вариант подходит именно вам.

---

[a] https://codeit.guru/en_US.

В приложении к этой книге приведены инструкции по установке и обновлению программного обеспечения для поддержки HTTP/2 некоторых распространенных веб-серверов и платформ. В зависимости от операционной системы и пользователей вашего веб-сервера процесс может быть довольно сложным. Со временем он станет проще, так как настройки по умолчанию будут обновлены до версий с поддержкой HTTP/2. Однако в ближайшие несколько лет многим из нас придется потрудиться для включения поддержки HTTP/2 в программное обеспечение своих веб-серверов.

Как вы увидите в следующих двух разделах, сегодня доступны и другие пути перехода на HTTP/2. При определенных настройках балансировщика нагрузки ваш сервер может не предоставлять HTTP/2 пользователям. Такое случается, когда сам балансировщик не поддерживает HTTP/2.

Если вы хотите настроить простой веб-сервер для экспериментов с HTTP/2 и, возможно, последовать некоторым примерам из этой книги, я рекомендую выбрать веб-сервер, который вам больше всего подходит. Если у вас нет особых предпочтений, Apache является наиболее полнофункциональным из популярных веб-серверов благодаря его доступности на многих платформах, а также поддержке HTTP/2 push и HTTP/2 прокси (о которых мы поговорим позже).

3.2.2 HTTP/2 с обратным прокси-сервером

Еще одним вариантом реализации HTTP/2 выступает установка *обратного прокси-сервера*, поддерживающего данную версию протокола; такие прокси могут переводить запросы на HTTP/1.1 и передавать их на ваш веб-сервер, как показано на рис. 3.5.

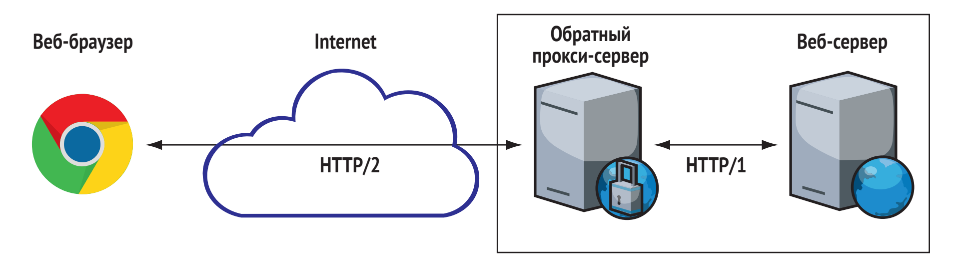


Рис. 3.5 Реализация HTTP/2 с обратным прокси

Обратные прокси-серверы, как следует из их названия, работают ровно наоборот относительно стандартных перехватывающих прокси. Последние «защищают» сеть от внешнего мира и прокладывают путь исходящему трафику, за счет чего и происходит связь с Internet. Обратные прокси обрабатывают входящий трафик и обеспечивают ему доступ к серверам, к которым нет прямого доступа. Такой тип прокси достаточно распространен и используется повсеместно обычно по одной из двух следующих причин:

- обратный прокси работает как *балансировщик нагрузки*;
- он облегчает работу HTTPS или HTTP/2.

Если обратный прокси используется в качестве балансировщика нагрузки, он работает как минимум с двумя веб-серверами и отправляет трафик на любой из них в зависимости от режима настройки (режим реального времени или режим ожидания). Балансировщики нагрузки, работающие в реальном времени, используют специальный алгоритм, который помогает им решить, как правильно разделить трафик (например, на основе исходного IP-адреса или алгоритма циклического перебора). На рис. 3.6 изображена реализация обратного прокси как балансировщика нагрузки.

Если вы используете обратный прокси именно таким образом, возможно, вам удастся перевести на HTTP/2 только балансировщик, при этом не затрагивая веб-сервер. Как упоминалось ранее, перевод веб-сервера на HTTP/2 может не дать положительного эффекта, если весь трафик сначала проходит через балансировщик нагрузки. Некоторые существующие балансировщики (такие как F5, Citrix Netscaler и HAProxy) уже поддерживают HTTP/2. Остальные также скоро добавят поддержку данной версии.

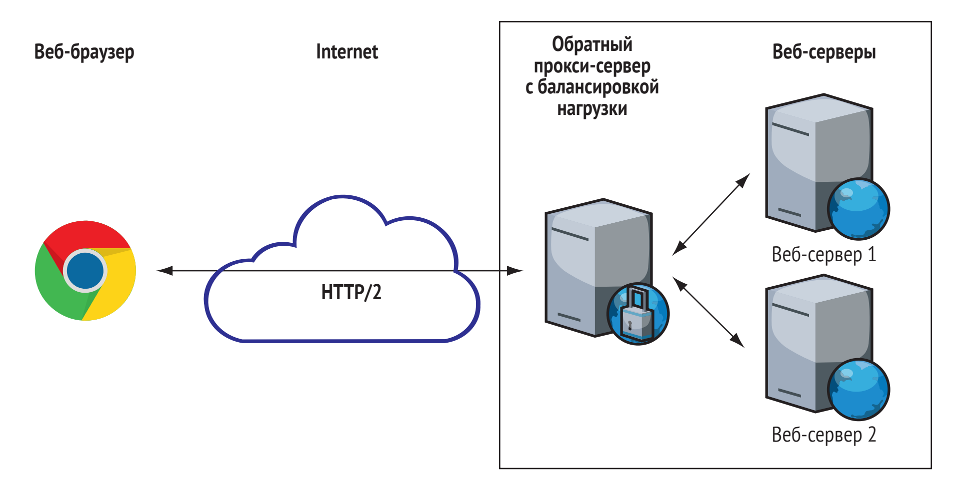


Рис. 3.6 **Обратный прокси-сервер как балансировщик нагрузки**

Необходимо ли использовать HTTP/2 постоянно?

При реализации HTTP/2 с обратным прокси происходит следующее: соединение проходит через HTTP/2 до момента контакта с обратным прокси, после чего открывается отдельное соединение (возможно, проходящее через HTTP/1.1). Этот процесс схож с тем, как при использовании того же обратного прокси, HTTPS-соединение переходит в HTTP после контакта с ним. Такой способ является распространенным вариантом упрощения настройки HTTPS, так как здесь сертификаты необходимо настраивать лишь в точке входа, где ими также можно управлять). Данный вариант пользуется популярностью по причине того, что в прошлом для установки HTTPS-соединений требовалось большое количество ресурсов (сейчас, благодаря увеличению вычислительной мощности, потребность в этом снизилась).

Итак, необходимо ли все же постоянно использовать HTTP/2, и что вы теряете при использовании HTTP/1.1-соединений на стороне сервера?

Основным преимуществом HTTP/2-соединений является высокая скорость, в сравнении с другими соединениями с высокой задержкой и низкой пропускной способностью. Такие характеристики могут быть присущи соединениям, с помощью которых пользователи подключаются к вашему пограничному серверу (в данном случае это обратный прокси). Здесь проблемы производительности HTTP/1.1 не так страшны, так как путь, который должен пройти трафик от вашего обратного прокси до остальной части веб-инфраструктуры, вероятнее всего, небольшой.

Использовать отдельное соединение для прохождения HTTP/2-трафика от обратных прокси к остальным серверам не совсем разумно, так как последние не ограничиваются шестью соединениями, установленными браузерами. Более того, существует мнение, что использование отдельного соединения может привести к проблемам с производительностью, однако это зависит

от того, как оно реализовано на обратном прокси и на конечном сервере. Отчасти по этой причине nginx заявил, что откажется от реализации HTTP/2 для прокси-соединений[a].

Таким образом, как и в случае с HTTPS, для базовой поддержки HTTP/2 нет необходимости использовать эту версию протокола на всех участках вашей инфраструктуры. Даже функции, работающие исключительно при поддержке HTTP/2, например HTTP/2 push, можно настроить и реализовать в такой системе. Об этом мы поговорим в главе 5.

[a] http://mailman.nginx.org/pipermail/nginx/2015-December/049445.html.

В других случаях, когда обратный прокси используется не в качестве балансировщика нагрузки, часто бывает, что веб-сервер (например, Apache или ngix) располагается вне внутренних серверов приложения (например, Tomcat или Node.js). Таким образом, веб-сервер передает некоторые запросы еще и внутреннему серверу, а не только прокси (см. рис. 3.7).

Такой способ имеет ряд преимуществ. Основным из них является то, что вы освобождаете веб-сервер от лишней загрузки, например статических ресурсов (изображений, CSS-файлов, библиотек JavaScript и т. д.). Кроме того, вы разгружаете работу HTTPS и, конечно, HTTP/2. Снижая нагрузку на сервер приложений, вы позволяете ему лучше выполнять свою работу: обрабатывать динамические ресурсы и выполнять поиск в базе данных.

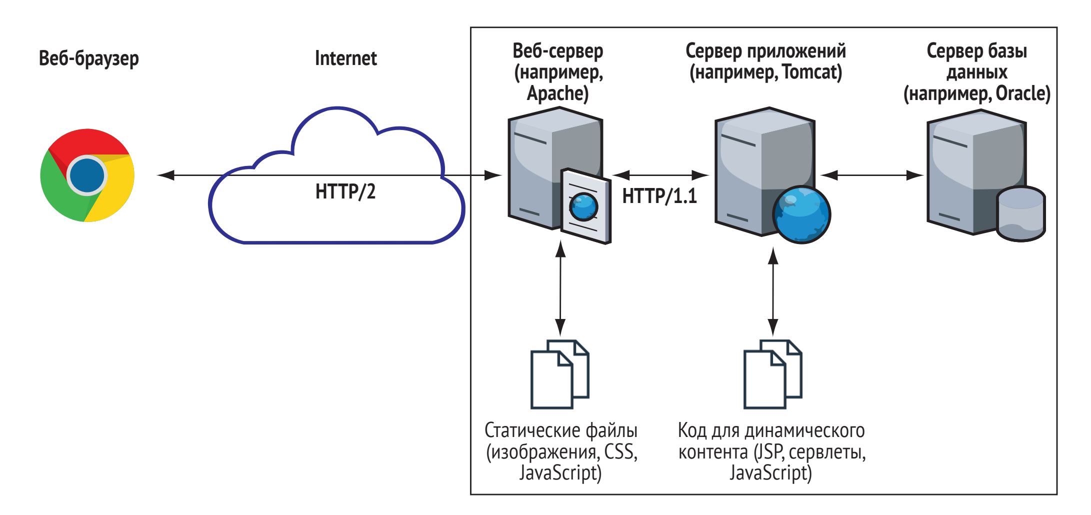


Рис. 3.7 **Веб сервер, располагающийся вне сервера приложений/сервера базы данных**

Кроме того, данный способ имеет преимущества с точки зрения безопасности, поскольку первой точкой контакта является веб-сервер, который в свою очередь может защитить более уязвимый сервер приложений

или сервер базы данных от вредоносных запросов. Поэтому поддержку HTTP/2 возможно реализовать и на сервере приложений: просто разместите перед ним другой сервер, поддерживающий HTTP/2.

Обратный прокси может помочь и при тестировании HTTP/2 и проверке его влияния на ваш сайт. Для этого следует разместить обратный прокси рядом с вашим сервером под другим именем (например, http2.example.com вместо test.example.com). Затем прокси сделает запрос к основному серверу, используя HTTP/1.1 через высокоскоростное локальное соединение, как изображено на рис. 3.8.

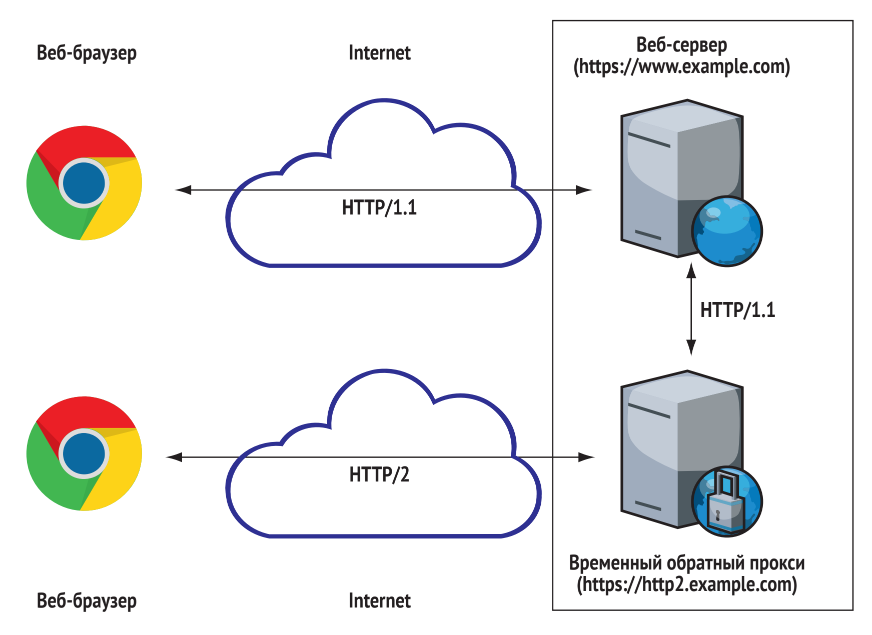


Рис. 3.8 Тестирование HTTP/2 с помощью временного добавления обратного прокси

Если ваш сервер уже поддерживает HTTP/2, но вы еще не переключились на него, вам может не потребоваться создание отдельного прокси-сервера. Если вы создадите виртуальный хост с выключенным HTTP/2 и отдельным именем, вы сможете запустить как сайты на HTTP/1.1, так и на HTTP/2; следовательно, вы сможете протестировать HTTP/2 перед его использованием на реальном хосте, которым пользуются посетители веб-сайта.

3.2.3 HTTP/2 и CDN

CDN – это всемирная сеть серверов, которые могут выступать локальной точкой контакта для вашего веб-сайта. Посетители сайта подключаются к ближайшему серверу CDN, так как сеть располагает большим количеством DNS-записей по всему миру. Запросы направляются на ваш сервер

(исходный сервер), а их копии кешируются в CDN, что ускоряет процесс передачи идентичного запроса в последующие разы. Большинство CDN-серверов уже поддерживает HTTP/2, поэтому вы можете воспользоваться ими, оставив при этом исходный сервер на HTTP/1.1. Такой способ схож с использованием обратного прокси. Однако отличаются они тем, что сети CDN имеют собственные обратные прокси и управляют ими вместо вас. Вариант реализации вышеописанного способа показан на рис. 3.9.

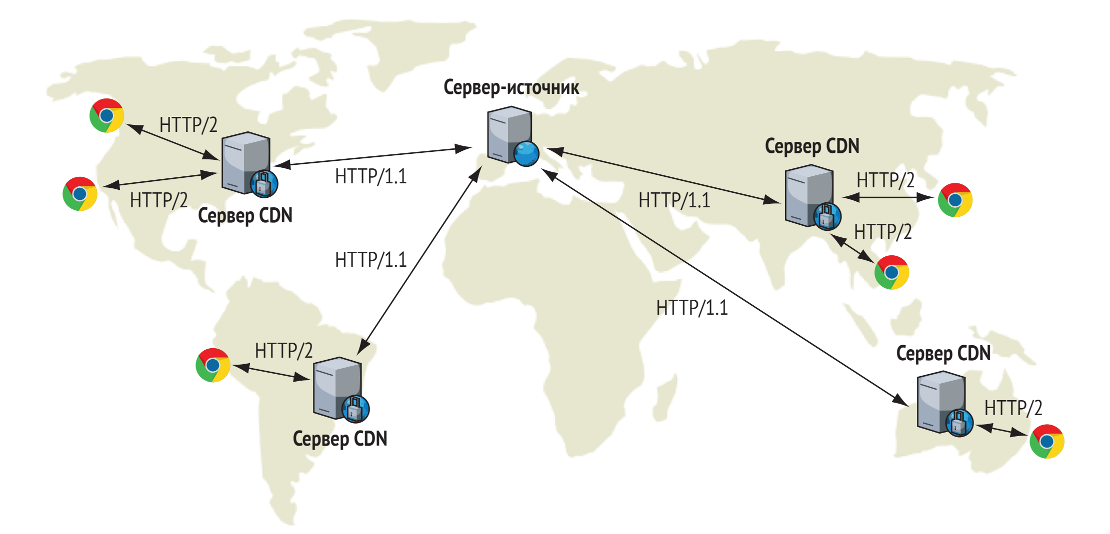


Рис. 3.9 **Переключение на HTTP/2 с помощью CDN**

Несмотря на то что при реализации данного способа вы используете дополнительные элементы инфраструктуры, подключение через CDN может быть значительно быстрее прямого, поскольку оно позволяет локальному серверу обрабатывать некоторые запросы настройки подключения для клиента (например, установка исходного TCP-соединения или согласование HTTPS). Преимущества такого подхода перекрывают недостатки наличия дополнительного серверного перехода при совершении запроса. Кроме того, CDN-серверы позволяют кешировать еще и ответы, таким образом, все дополнительные запросы обрабатываются локальным сервером, а не исходным. Таким образом, экономится время, снижается нагрузка на исходный сервер и увеличивается его пропускная способность.

CDN – это форсированные обратные прокси-серверы. До появления HTTP/2 их использовали преимущественно для повышения производительности, но, как оказалось, они также могут облегчить переключение сервера на новую версию протокола.

CDN могут работать и с HTTPS-подключениями, необходимыми для HTTP/2. Однако, если исходный сервер не поддерживает HTTPS, то трафик будет шифроваться только до контакта с ним. В основном HTTPS используют для снижения рисков на клиентской стороне (например,

подключение к неизвестной сети Wi-Fi сопряжено с рисками, которые в силах решить HTTPS), но предпочтительнее все же использовать HTTPS от начала и до конца соединения. Многие люди считают, что недобросовестно выгружать HTTPS в CDN, если весь оставшийся путь будет проходит через HTTP-соединение, так как в таком случае посетители веб-сайта не будут осведомлены о том, что их пароли потенциально передаются по незащищенному соединению. Настроить HTPPS в соответствии со всеми требованиями HTTP/2 (например, ALPN) непросто. Однако, по крайней мере, вы можете использовать для этой цели CDN, а для подключения к исходному серверу использовать старую конфигурацию HTTPS через HTTP/1.1.

Сети CND дают нам множество преимуществ, в том числе они могут обеспечить поддержку HTTP/2. Они давно внедрили поддержку HTTP/2, а некоторые CND предоставляют доступ к их функционалу бесплатно, что будет полезно для небольших сайтов. CDN также могут выполнять расшифровку трафика, поэтому работать с такой третьей стороной вам будет довольно комфортно.

3.2.4 Вывод по реализации HTTP/2

Существует несколько способов реализации HTTP/2 на вашем сайте. Выбор нужного вам способа зависит от того, какую инфраструктуру для обработки трафика вы используете. К сожалению, прямое подключение HTTP/2 – это довольно сложно, а также требует большого количества настроек, которые необходимо выполнить вручную. По мере распространения данной версии и обновления серверного программного обеспечения ситуация будет меняться. Однако сейчас это все еще трудный процесс.

Помимо прямого подключения существую и другие варианты. Например, использование обратных прокси или CDN. Данные методы упрощают переход на HTTP/2, пока поддержка новой версии распространена не на всех широко известных серверах.

Сегодня вы можете выбрать способ, который лучше всего подходит для вашего веб-сайта, а также провести некоторые эксперименты, которые покажут, как работает выбранный вами способ. Остальная часть нашей книги будет полезна для тех, кто уже имеет серверы с поддержкой HTTP/2, на которых они смогут опробовать ряд примеров. Однако некоторые примеры будут приведены и на общедоступных веб-сайтах.

3.3 Устранение неполадок при настройке HTTP/2

Вы можете узнать, используется ли HTTP/2, заглянув в инструменты разработчика в вашем браузере. Иногда, несмотря на то что сервер использует HTTP/2, бывает сложно запустить работу протокола ввиду ряда причин, упомянутых в этой главе. Рассмотрим некоторые из них.

- *HTTP/2 не поддерживается на вашем сервере*. Очевидно, что для использования HTTP/2 ваш сервер должен его поддерживать. Как уже было сказано, на сегодняшний день большинство серверов не поддерживает HTTP/2 по умолчанию. Вам следует проверить, какую версию серверного программного обеспечения вы используете и была ли в нее добавлена поддержка HTTP/2. Обратите внимание, что установка последних обновлений (например, с помощью yum update или apt-get) не гарантирует того, что ваш сервер начнет поддерживать HTTP/2.
- *HTTP/2 не включен на вашем сервере*. Даже если сервер поддерживает HTTP/2, использование этой версии может быть отключено. Некоторые серверы (например, IIS) по умолчанию настроены на HTTP/2. На других серверах (например, Apache) поддержка HTTP зависит от используемой конфигурации или сборки. Например, сборки ApacheHaus для Windows включают HTTP/2 по умолчанию, но при установке из исходного кода такая функция не работает. Кроме того, начиная с версии 2.4.27, Apache не поддерживает HTTP/2 при использовании модуля `prefork mpm`[1].

  Также некоторые параметры компиляции (например, `--enable-http2` для Apache и `--withhttp_v2_module` для nginx) необходимы для включения HTTP/2, но в свою очередь не включают его по умолчанию. Если HTTP/2 не работает на вашем сервере, посмотрите соответствующие инструкции в документации.
- *На вашем сервере не включен HTTPS*. Как вы уже знаете из раздела 3.1.1, веб-браузеры поддерживают HTTP/2 только через HTTPS-соединения. Если ваш сайт использует HTTP-соединение, вы не сможете включить HTTP/2.
- *На вашем веб-сервере не включена поддержка ALPN*. ALPN – это расширение протокола TLS, на основе которого создается сеанс HTTPS, позволяющий использовать HTTP/2. Некоторые браузеры (например, Safari, Edge и Internet Explorer на момент написания данной книги) позволяют перейти на HTTP/2 и при использовании более старого NPN, так и при использовании более нового ALPN. Другие же браузеры (например, Chrome, Firefox и Opera) требуют использования именно ALPN.

  Вы можете проверить поддержку ALPN с помощью онлайн-инструментов, таких как SSLLabs[2] (который запускает комплексный тест для настройки HTTPS, однако его проведение занимает несколько минут) или KeyCDN HTTP/2 Test[3] (который работает быстрее, поскольку тестирует только HTTP/2 и ALPN). Если ваш сервер не является общедоступным, вы не можете использовать вышеописанные инструменты, зато можете пользоваться инструмента-

1 https://github.com/icing/mod_h2/releases/tag/v1.10.7.
2 https://www.ssllabs.com/ssltest/.
3 https://tools.keycdn.com/http2-test.

ми командной строки, такими как s_client OpenSSL (при условии, что ваша версия OpenSSL поддерживает ALPN):

```
openssl s_client -alpn h2 -connect www.example.com:443 -status
```

В качестве альтернативы вы можете использовать инструмент testssl[1], который способен выполнить большинство тестов SSLLabs. Однако для полного тестирования поддержки HTTP/2 все же требуется версия OpenSSL с поддержкой ALPN.

Как и браузеры, некоторые веб-серверы (например, Apache) используют исключительно ALPN; другие (например, nginx) поддерживают и ALPN, и NPN. Поддерживает ли ваш сервер ALPN, зависит от версии библиотеки TLS, которую вы используете. В табл. 3.3 приведен список библиотек, поддерживающих ALPN. Если вы не знаете, какую библиотеку TLS используете, скорее всего, это будет OpenSSL для Linux, LibreSSL для macOS или SChannel для Windows.

Таблица 3.3 Поддержка ALPN TLS-библиотеками

Библиотека TLS	Версия с поддержкой ALPN
OpenSSL	1.0.2
LibreSSL	2.5.0
SChannel (использовалась приложениями Microsoft)	8.1/ 2012 R2
GnuTLS	3.2.0

Даже если ваша библиотека TLS поддерживает ALPN, ваше серверное программное обеспечение могло быть создано с другой ее версией. Например, в RHEL/CentOS 7.4 используется OpenSSL 1.0.2, но версии Apache и nginx, установленные по умолчанию, собраны еще с применением OpenSSL 1.0.1, поэтому они не поддерживают ALPN. При перезапуске Apache обычно добавляет строку в журнал ошибок, в которой указана версия OpenSSL:

```
[mpm_worker:notice] [pid 19678:tid 140217081968512] AH00292:
Apache/2.4.27 (Unix) OpenSSL/1.0.2k-fips configured -- resuming normal
operations
```

В качестве альтернативы вы можете добавить `ldd` перед `mod_ssl`:

```
$ ldd /usr/local/apache2/modules/mod_ssl.so | grep libssl
libssl.so.10 => /lib64/libssl.so.10 (0x00007f185b829000)
$ ls -la /lib64/libssl.so.10
lrwxrwxrwx. 1 root root 16 Oct 15 16:07 /lib64/libssl.so.10 ->
libssl.so.1.0.2k
```

Для того чтобы узнать сборку ngix, вы можете использовать параметр -V:

```
$ nginx -V
nginx version: nginx/1.13.6
built by gcc 4.8.5 20150623 (Red Hat 4.8.5-16) (GCC)
```

[1] https://testssl.sh/.

```
built with OpenSSL 1.0.2k-fips 26 Jan 2017
TLS SNI support enabled
configure arguments: --with-http_ssl_module --with-http_v2_module
```

Способы для других серверов прописаны в их документации.

- *На вашем веб-сервере не поддерживаются надежные шифры HTTPS.* В спецификации HTTP/2 перечислен ряд шифров, которые клиент не должен использовать для HTTP/2-соединения[1]. Некоторые браузеры (например, Chrome) не используют их, поэтому, если вы хотите перейти на HTTP/2, вам следует пользоваться такими шифрами, как ECDHE GCM или POLY, которые на момент написания данной книги являются наиболее подходящими. В большинство реализаций по умолчанию включены более надежные шифры, однако при переносе старой конфигурации шифра из предыдущей реализации они могут быть не включены.

  Настроить шифр можно с помощью инструмента онлайн-тестирования SSLLabs. Поначалу вам, возможно, будет сложно в нем разобраться, однако именно он способен дать полную информацию о настройке HTTPS на вашем сервере, а также указать, поддерживается ли HTTP/2 для обычных клиентов.

  Также вы можете воспользоваться ресурсом Mozilla SSL Configuration Generator[2]. Большинство сайтов должно использовать конфигурацию Modern, хотя для старых клиентов подойдет и Intermediate.
- *Используется перехватывающий прокси, который переводит вас на HTTP/1.1.* Прокси (например, в корпоративной среде) или антивирусное программное обеспечение могут переводить вас на HTTP/1.1, поскольку они перехватывают HTTPS-соединение. Мы обсуждали это в разделе 3.1.1. Очень важно удостовериться, что сертификат HTTPS для вашего веб-сайта был выдан настоящим центром сертификации.

  Если ваш сайт является общедоступным, такие инструменты, как SSLLabs или KeyCDN HTTP/2 Test[3] помогут узнать, поддерживает ли он HTTP/2. Если да, то проблема может носить локальный характер и, возможно, она может быть вызвана перехватывающим прокси.

  Также перехватывать HTTPS или добавлять некоторые сайты в список разрешенных могут различные вредоносные программы.
- *Заголовок обновления был перенаправлен неверно*. Внутренний сервер (например, Apache) может использовать заголовок `Upgrade: h2` при переключении на HTTP/2. Если этот заголовок неверно перенаправляется прокси-сервером, могут возникнуть проблемы. Браузер пытается перейти на HTTP/2 (как следует из заголовка) и терпит неудачу, поскольку обратный прокси-сервер не поддерживает HTTP/2. В таком случае прокси не должен отправлять заголовок `Upgrade`. В главе 4 мы обсудим эту проблему подробнее. Браузер Safari

1 https://httpwg.org/specs/rfc7540.html#BadCipherSuites.
2 https://mozilla.github.io/server-side-tls/ssl-config-generator/.
3 https://tools.keycdn.com/http2-test.

подвержен возникновению подобных ситуаций и зачастую выдает ошибку `nsposixerrordomain:100`.
- *Заголовки HTTPS недействительны*. При наличии недействительных заголовков (если, например, в них присутствуют пробелы или двойные двоеточия) браузер Chrome отправляет сообщение `ERR_SPDY_PROTOCOL_ERROR`[1]. К слову, при использовании HTTP/1.1 он прощает подобные недочеты. Safari в такой ситуации может выдать уже знакомую нам ошибку `nsposixerrordomain: 100`.
- *В кеш попадают элементы исходного протокола загрузки*. Если вы обновили сервер до поддержки HTTP/2 и пытаетесь его протестировать, не очистив предварительно кеш, в тесте будут использованы кешированные ресурсы. Кешированные элементы показывают версию HTTP, которая использовалась для загрузки запроса (это может быть и HTTP/1.1). В таком случае браузер отправит ответ `304 Not Modified`.

Резюме

- Почти все основные браузеры предоставляют поддержку HTTP/2 на стороне клиента.
- Поддержка HTTP/2 на стороне сервера доступна в более новых версиях, но зачастую эти версии нелегко установить без полного обновления сервера и/или вручную.
- Доступны различные варианты реализации поддержки HTTP/2, в том числе и посредством использования сторонней инфраструктуры, такой как CDN.
- Существует ряд причин, по которым HTTP/2 не может использоваться даже после того, как он был включен.

[1] https://www.michalspacek.com/chrome-err_spdy_protocol_error-and-an-invalid-http-header.

Часть II

Использование HTTP/2

В первой части этой книги я рассказал о необходимости и преимуществах перехода на новую версию HTTP, описал сущность HTTP/2, а также представил способы его настройки для вашего веб-сайта. Большинство людей захочет перейти на HTTP/2 уже на этом этапе. HTTP/2 разрабатывался так, чтобы переход на него был довольно простым, а сразу после перехода пользователь мог ощутить все преимущества новой версии.

Однако, чтобы ощутить эти преимущества в полной мере, предпочтительнее разобраться в более глубинных частях работы протокола. В этой части книги описаны основные аспекты новой версии. В главах 4 и 5 рассматриваются технические детали протокола, благодаря которым владельцы веб-сайтов и разработчики смогут извлекать из HTTP/2 как можно больше преимуществ. В главе 6 я несколько отойду от самой сущности протокола и расскажу о его значении для веб-производительности, а также рассмотрю возможное изменение методов, которое позволит оптимизировать их для мира HTTP/2.

4 Основы протокола HTTP/2

В этой главе мы рассмотрим:

- основы протокола HTTP/2: что это такое и чем он отличается от HTTP/1.1;
- переход на HTTP/2 со стороны клиента и сервера;
- фреймы HTTP/2 и их отладку.

В этой главе мы поговорим об основах HTTP/2 (фреймы, потоки и мультиплексирование). Более сложные части протокола (такие как приоритет потоков и управление ими) мы обсудим в главах 7 и 8. Исчерпывающая информация о протоколе содержится в спецификации HTTP/2[1]. Во время или после прочтения данной главы вы можете обращаться к этому документу. Однако в главе 4 я обращаю ваше внимание на дополнительные детали и привожу интересные примеры, которые (я надеюсь) упростят процесс изучения протокола.

4.1 Почему HTTP/2, а не HTTP/1.2?

В главе 2 мы обсудили различия между HTTP/1 и HTTP/2. Новая версия протокола была создана специально для решения проблем с производительностью, которые возникали при использовании старой версии. В новую версию были внесены следующие изменения:

---

[1] https://tools.ietf.org/html/rfc7540.

- протокол стал двоичным, а не текстовым;
- передача данных осуществляется мультиплексированно, а не синхронно;
- добавлено управление потоками информации;
- стала возможна установка приоритета потоков;
- появилась возможность сжатия заголовков;
- реализована технология server push.

Все вышеперечисленные изменения (которые мы позже обсудим более подробно) являются фундаментальными и не имеют обратной совместимости. В то время как веб-сервер, работающий на HTTP/1.0, мог принимать сообщения от серверов HTTP/1.1, игнорируя при этом новый функционал, перешедшие на HTTP/2 серверы используют уже совсем иную структуру и формат данных и поэтому не могут принимать запросы старых версий. Именно по этой причине протокол HTTP/2 считается базовым обновлением версии.

Различия между старой и новой версией представлены в основном на уровне отправки и получения сообщений сервером и клиентом. На других уровнях, с которыми сталкивается большинство разработчиков (например, семантика HTTP)[1], работа новой версии во многом схожа со старой. У протокола HTTP/2 можно наблюдать уже знакомые методы (`GET`, `POST`, `PUT` и т. д.), URL-адреса, коды ответа (`200`, `404`, `301`, `302`) и HTTP-заголовки (большинство). Однако HTTP/2 выполняет отправку все тех же HTTP-запросов намного эффективнее.

Во многих аспектах HTTP/2 похож на HTTPS. Он так же эффективно «обертывает» стандартные HTTP-сообщения в специальный формат перед отправкой и «разворачивает» их после получения. Таким образом, несмотря на то что клиент (веб-браузер) и сервер (веб-сервер) требуют полного соответствия версий на уровне протокола, обработка сообщений данных версий на более высоких уровнях происходит практически одинаково, поскольку они используют одни и те же базовые концепции HTTP. Однако разработка сайтов с помощью HTTPS и HTTP/2 отличается. Точно так же, как всестороннее изучение HTTP/1 позволило разработчикам оптимизировать многие сайты (как уже обсуждалось в главе 2), исследование HTTP/2 поможет разработать новые варианты оптимизации, повысить эффективность работы веб-разработчиков и увеличить скорость работы сайтов. Именно по этой причине важно понять, какие новшества были введены в новую версию протокола.

HTTP/2.0 или HTTP/2?

Изначально HTTP/2 назывался HTTP/2.0, но впоследствии рабочая группа HTTP решила отказаться от дополнительного номера версии (.0). Название HTTP/2 характеризует все те новшества, которые были добавлены в него (такие как двоичный формат, мультиплексирование и т. д.). Разработчики предполагают, что новые версии (например, HTTP/2.1) и их реализации бу-

[1] https://tools.ietf.org/html/draft-ietf-httpbis-semantics.

дут совместимы с основным протоколом. Похожая ситуация была и с HTTP/1 (данное название не прижилось, однако в книге мы используем его для обозначения версий HTTP/1.0 и HTTP/1.1), протоколом, имеющим текстовый формат, структура которого состоит из заголовков и тел.

Кроме того, в сообщениях HTTP/2, в отличие от формата сообщений предыдущих версий, не указывается номер версии. Например, в HTTP/2 не используются запросы наподобие `GET /index.html`. Однако дополнительный номер версии часто фигурирует в файлах журналов некоторых реализаций. Например, в файлах журнала Apache при обработке запросов в формате HTTP/1 отображается номер версии HTTP/2.0:

```
78.1.23.123 - - [14/Jan/2018:15:04:45 +0000] 2 "GET / HTTP/2.0" 200 1797
"-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36"
```

Таким образом, в журнале вы видите сообщение типа HTTP/1 (несмотря на утверждение об обратном выше). Но на самом деле этот запрос не настоящий, он искусственно создается сервером в целях упрощения обработки журнальных сообщений. Фактически название HTTP/2.0 встречается только в предисловии спецификации протокола (см. раздел 4.2.5).

4.1.1 Двоичный, а не текстовый

Одно из основных различий между HTTP/1 и HTTP/2 состоит в том, что последний является двоичным протоколом на основе пакетов, в то время как первый полностью основан на тексте. С текстовыми протоколами проще работать людям, однако компьютерам их обрабатывать сложнее. Изначально HTTP был простым протоколом запроса–ответа, и текстовый формат был для него приемлем, однако такой формат все больше ограничивает его использование в современной сети Internet.

Текстовый формат предполагает последовательную отправку запросов, т. е. перед отправкой нового запроса предыдущий должен быть полностью завершен, и на него должен быть получен ответ. Именно так и работал HTTP на протяжении последних 20 лет, хотя в него были внесены небольшие улучшения. Например, в HTTP/1.0 ввели двоичный формат тела ответа, благодаря чему ответы могли содержать изображения и другие медиафайлы, а в HTTP/1.1 впервые появилась конвейеризация запросов (см. главу 2) и фрагментированная передача сообщений неопределенной длины. Последнее позволило отправлять содержимое тел запросов по частям, например сначала одну часть, а затем – по мере доступности – остальное содержимое. Таким образом, клиент, получивший фрагментированный ответ (или сервер, принимающий фрагментированный запрос), может начать обработку полученных частей, не дожидаясь остальных. Такую технологию часто используют в случаях, когда объем динамически сгенерированных данных заранее неизвестен. И фрагментированная передача, и конвейеризация подвержены проблеме блокировки заголовка (HOL), в результате которой сообщение из верхней части очереди прерывает отправку последующих ответов. Стоит

также отметить, что на практике крайне мало веб-серверов поддерживает конвейеризацию.

HTTP/2 полностью перешел на двоичный формат, где HTTP-сообщения разделяются и отправляются в отдельных фреймах. При использовании HTTP/2 все сообщения по умолчанию отправляются по частям. Спецификация HTTP/2 гласит:

> *Механизм фрагментированной передачи данных, определенный в разделе 4.1 RFC7230, НЕ ДОЛЖЕН использоваться в HTTP/2.*

Вышеупомянутые фреймы похожи на TCP-пакеты, на которых основывается большинство HTTP-соединений. Полное HTTP-сообщение складывается из всех полученных фреймов. Несмотря на то что HTTP/2 во многом похож на TCP, он обычно работает поверх TCP, а не заменяет его (хотя в Google, экспериментируя с заменой TCP на QUIC, создали более простую реализацию, где HTTP/2 работает поверх него, о чем мы поговорим в главе 9). TCP – это базовый протокол, который гарантирует доставку и правильный порядок сообщений, для чего он и используется в HTTP/2.

Двоичный формат HTTP/2 предназначен для отправки и получения сообщений, в то время как сами сообщения похожи на сообщения формата HTTP/1. Двоичные фреймы обычно обрабатываются клиентами нижнего уровня или библиотеками (браузеров или серверов). Как уже было сказано ранее, приложения более высокого уровня, такие как JavaScript, не заботятся о том, как именно отправляются сообщения. Поэтому они обрабатывают HTTP/2-соединение практически так же, как и HTTP/1.1. Однако понимание сути фреймов HTTP/2, а также их просмотр помогает при отладке неожиданных ошибок. Особенно актуален этот аспект на ранней стадии внедрения протокола, так как в это время могут возникать (надеемся, что редко!) проблемы реализации при определенных сценариях.

4.1.2 Мультиплексирование вместо синхронности

HTTP/1.1 являлся синхронным протоколом, позволяющим отправлять запросы (и получать на них ответы) по одному: клиент отправляет сообщение, а сервер получает ответ. В главе 2 мы уже обсудили, почему данный протокол неэффективен применимо к современной Всемирной паутине, где веб-сайты могут состоять из сотен ресурсов. Для подобных проблем были созданы обходные пути, в HTTP/1 решение представляло собой создание сразу нескольких соединений или отправку меньшего количества объемных запросов вместо множества маленьких. Однако оба этих пути оказались неэффективными, так как создавали новые проблемы. На рис. 4.1 показано создание трех TCP-соединений для параллельной отправки и приема трех запросов HTTP/1. Обратите внимание, что запрос 1 не отображается на главной странице, так как он является начальным, и только после его завершения в запросах 2–4 может быть параллельно запрошено несколько ресурсов.

HTTP/2 позволяет выполнять несколько запросов одновременно в одном соединении, и для этого он использует разные потоки для каждого HTTP-запроса или ответа. Такая технология стала возможной благодаря переходу на использование двоичных фреймов, где каждый фрейм имеет свой идентификатор потока. Принимающая сторона сможет восстановить сообщение целиком, после того как все фреймы для данного потока будут получены.

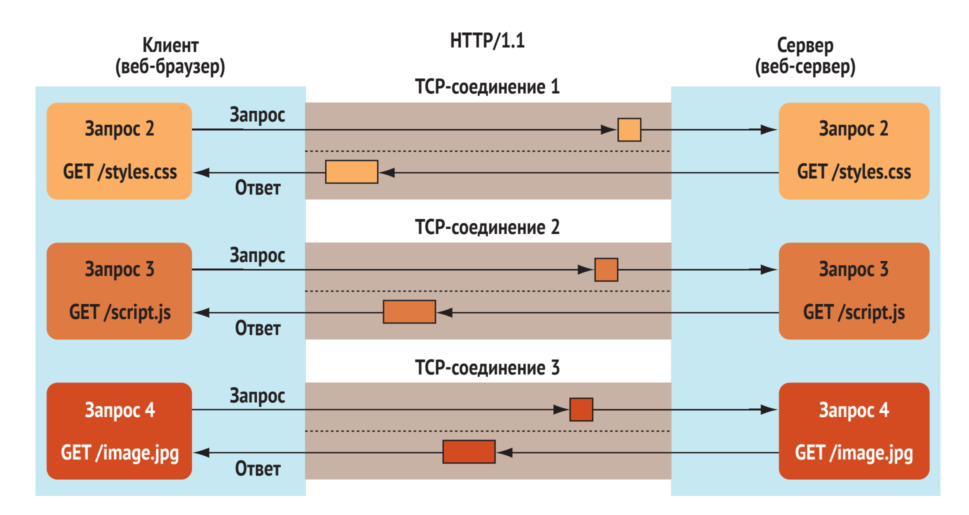


Рис. 4.1 Параллельное выполнение нескольких запросов HTTP/1 требует нескольких TCP-соединений

Фреймы – это ключ к возможности параллельной отправки нескольких сообщений. Каждый фрейм содержит метку, которая указывает какому сообщению (потоку) он принадлежит. Таким образом, вы можете отправлять или получать один, два, три или даже сто сообщений одновременно в одном мультиплексном соединении, в то время как при использовании HTTP/1 большинство браузеров ограничивалось шестью. На рис. 4.2 также показаны три запроса, но, в отличие от 4.1, они отправляются последовательно в одном соединении (аналогично, как при конвейеризации HTTP/1.1), а ответы отправляются обратно в смешанном виде (что уже невозможно при конвейеризации).

На данном примере мы видим, что запросы отправляются в разное время, поскольку в конечном итоге каждый фрейм должен отправляться по одному и тому же HTTP/TCP-соединению после предыдущего. Так же происходит и при использовании HTTP/1.1, так как для запросов, хотя они и кажутся параллельными, существует только одно сетевое соединение, и поэтому все они помещаются в очередь для отправки на сетевом уровне. Основное отличие заключается в том, что соединение HTTP/2 не блокируется после отправки запроса до того момента, пока на него не будет получен ответ (в HTTP/1.1, как было описано в главе 2, соединение закрывается).

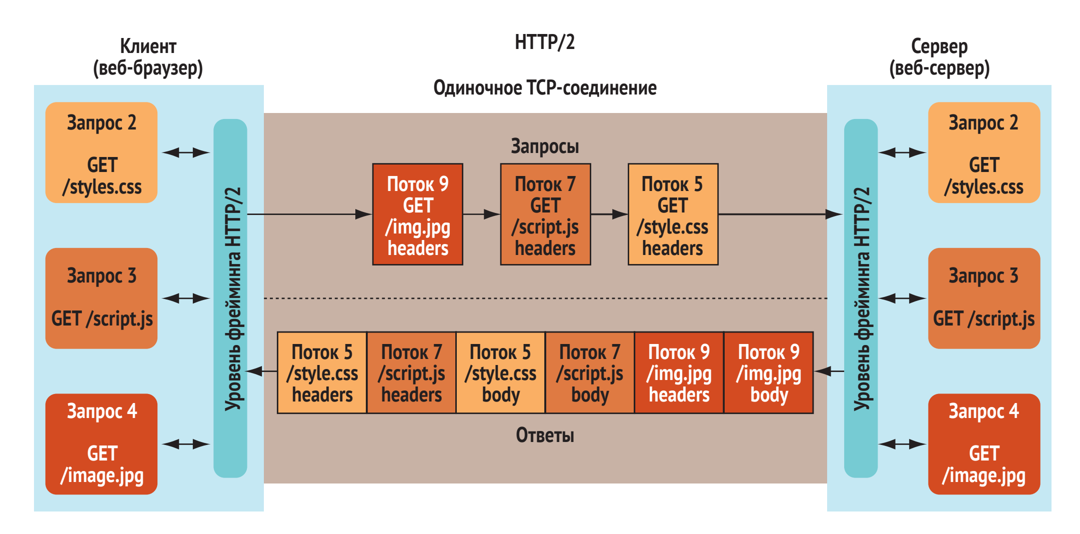


Рис. 4.2 Запрос трех ресурсов с помощью мультиплексного соединения HTTP/2

Точно так же ответы могут быть отправлены обратно в смешанном виде (потоки 5 и 7 на рис. 4.2) или последовательно (поток 9 на рис. 4.2). Порядок, согласно которому сервер отправляет ответы, полностью определяется на серверной стороне, однако клиент все же может указывать приоритетные потоки. Если существует возможность отправки сразу нескольких ответов, сервер может отдавать приоритет важным ресурсам (таким как CSS и JavaScript), а к второстепенным относить другие ресурсы (такие как изображения). Подробнее об этом мы поговорим в главе 7.

Каждый запрос получает новый *инкрементный идентификатор потока* (*ID*) (5, 7 и 9 на рис. 4.2), в соответствии с которым позже отправляются ответы. Таким образом, потоки являются двунаправленными, как и сами HTTP-соединения. После получения ответа потоки закрываются. Поток HTTP/2 не является прямым аналогом соединения HTTP/1.1, так как при использовании новой версии протокола они отбрасываются и не используются повторно, в то время как в HTTP/1.1-соединение остается открытым и его можно использовать для отправки другого запроса.

В целях предотвращения конфликтов идентификаторов потоков была придумана следующая система: запросам, инициированным клиентом, присваиваются нечетные номера (например, 5, 7 и 9 на наших рисунках), а запросам, инициированным сервером, – четные. Обратите внимание, что на момент написания этой книги, серверы не обладают технической возможностью инициировать потоки, за исключением определенных случаев, где в конечном итоге они являются ответом на поток клиента. Об этом мы поговорим в главе 5. Как упоминалось ранее, ответы помечаются идентификаторами, соответствующими запросам. Идентификатор потока 0 (не показан на рисунках) – это поток управления соединением, который может использовать как клиент, так и сервер.

Понимание рис. 4.2 – ключ к пониманию HTTP/2. Если вы поняли эту концепцию и ее отличия от HTTP/1, вы прошли долгий путь изучения

HTTP/1. Конечно же, существуют некоторые сложности и тонкости, но на данном рисунке можно увидеть две главные особенности протокола HTTP/2:

- для отправки HTTP-запросов и ответов через одно TCP-соединение HTTP/2 создает несколько двоичных фреймов и использует мультиплексированные потоки;
- HTTP/2 отличается от HTTP/1 на уровне отправки сообщений, а на более высоком уровне основные концепции HTTP остаются неизменны. Запросы все также содержат метод (например, `GET`), информацию о необходимом ресурсе (например, `/styles.css`), заголовки, тело, коды состояния (например, `200`, `404`), кешированные куки-файлы и т. д.

Первый пункт означает, что работа HTTP/2 может быть изображена как на рис. 4.3, где каждый поток ведет себя как отдельное соединение HTTP/1. Однако данный рисунок может быть неверно истолкован теми, кто использует HTTP/1, поскольку потоки HTTP/2 не используются повторно (как соединения в HTTP/1), но и не являются полностью независимыми, и, как вы увидите в главе 7, это сделано не зря.

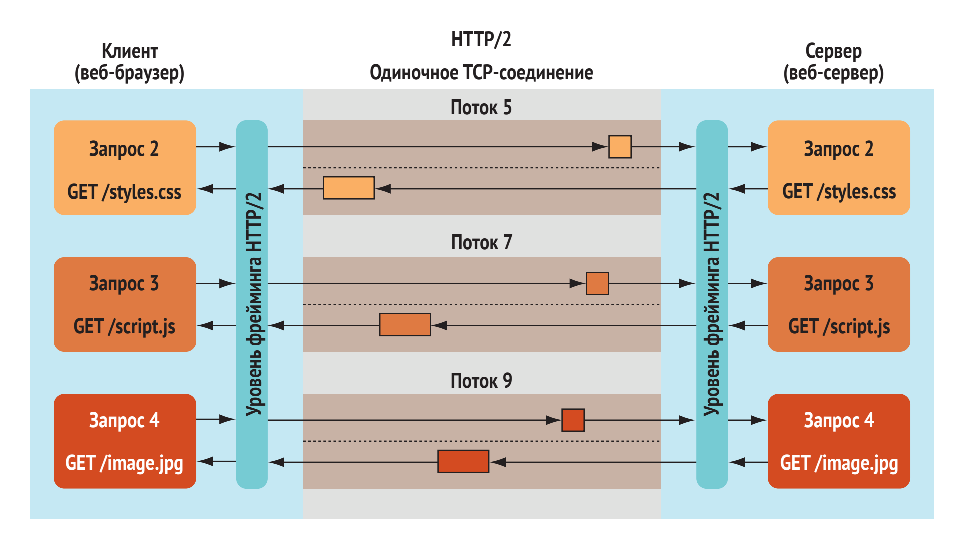


Рис. 4.3 Потоки HTTP/2 аналогичны соединениям HTTP/1

Второй пункт поясняет, почему HTTP/2 все еще не получил широкого распространения. Веб-браузеры и веб-серверы могут обрабатывать содержимое HTTP/2 на низких уровнях, поэтому пользователи (и разработчики) не задумываются о его работе и относятся к нему как к обычному протоколу. В разделе 4.2 указано, что HTTP/2 даже не нуждается в новой схеме (часть `https://` в начале URL-адреса). Фактически большинство людей уже использует HTTP/2 и даже не подозревает об этом. Данным протоколом пользуются Google, Twitter и Facebook, поэтому, если вы посещали данные сайты, вы тоже пользовались им.

Глубокое понимание принципа работы HTTP/1 позволило веб-разработчикам создавать более качественные сайты с высокой производительностью. Точно так же те из вас, кто уделит достаточно времени изучению HTTP/2, смогут пользоваться всеми преимуществами данного протокола.

4.1.3 Приоритет потоков и управление ими

До появления HTTP/2 протокол HTTP был простым протоколом, позволяющим в одном соединении отправлять один запрос и получать один ответ, поэтому в установке приоритетов не было необходимости. Клиент (обычно веб-браузер) определял приоритет и порядок отправки сообщений вне протокола HTTP. Кроме того, он использовал ограниченное количество HTTP/1-соединений (обычно шесть). Согласно такому принципу высоким приоритетом обладали критические ресурсы (блокирующие рендеринг элементы HTML, CSS и критический код JavaScript), а другие элементы, не блокирующие рендеринг (такие как изображения и асинхронный JavaScript), загружались позже. Запросы образовывали очередь и ожидали свободного HTTP/1-соединения, а порядок этой очереди, устанавливаемый браузером, определял приоритет.

HTTP/2 же обладает куда более высоким лимитом количества одновременно выполняемых запросов (во многих реализациях лимит по умолчанию составляет 100 активных потоков). Таким образом, браузеру уже не нужно составлять очередь из запросов, ведь большинство из них может быть отправлено в одно и то же время. Однако это может привести к снижению пропускной способности при загрузке ресурсов с более низким приоритетом (например, изображений), вследствие чего может казаться, что при использовании HTTP/2 страница загружается даже медленнее. Установка приоритетов потоков необходима для того, чтобы в первую очередь отправлялись наиболее важные ресурсы. Установка приоритетов реализуется сервером, который, в то время как очередь фреймов ожидает отправки, отправляет больше фреймов для запросов с высоким приоритетом, чем для запросов с низким приоритетом. Кроме того, приоритизация обеспечивает большую степень контроля потоков, чем в HTTP/1, где все соединения были независимы друг от друга. В HTTP/1, если не считать, что соединение может быть указано как не подлежащее использованию, отсутствовала возможность установки приоритетов. Например, если вам необходимо отправить пять критических ресурсов и один некритический, то при использовании HTTP/1 критические ресурсы могут быть отправлены по шести отдельным соединениям с одинаковым приоритетом, а шестой – задержан. При использовании HTTP/2 все шесть запросов могут быть отправлены с соответствующим приоритетом, и таким образом распределяется количество ресурсов для отправки каждого ответа.

Необходимость управления потоками – это еще одно следствие использования нескольких потоков в одном и том же соединении. И этот процесс крайне важен. Если получатель не может обрабатывать входя-

щие сообщения так же быстро, как отправитель, часть работы остается невыполненной, и эта часть должна буферизироваться, что в конечном итоге приводит к тому, что некоторые пакеты требуют повторной отправки. TCP позволяет контролировать пропускную способность соединения, однако HTTP/2 требует того же на уровне потока. Возьмем, к примеру, веб-страницу, на которой размещаются видеоматериалы прямой трансляции. Если пользователь останавливает видео, то было бы разумно приостановить загрузку данного потока, но при этом разрешить другим ресурсам продолжать загрузку через другие потоки.

В главе 7 мы еще раз затронем тему приоритизации потоков и управления ими, а также расскажем чуть подробнее о самих потоках. Мы поговорим об этом позже, так как обычно эти процессы контролируются браузером и сервером, поэтому пользователи и веб-разработчики практически не касаются их (на момент написания книги).

4.1.4 Сжатие заголовков

Посредством HTTP-заголовков отправляется дополнительная информация о запросах и ответах от клиента к серверу и обратно. Зачастую они могут повторяться. Рассмотрим некоторые заголовки, которые отправляются с каждым запросом и часто дублируют предыдущие:

- `Cookie`; куки-файлы отправляются с каждым запросом, поступающим в домен (за исключением нестандартных запросов, которые использует Amazon (см. главу 2), однако они, скорее, относятся к исключениям). Заголовки куки-файлов зачастую необходимы только для HTML-ресурсов, однако при этом они отправляются для каждого ресурса и могут быть довольно объемными;
- `User-Agent`; в данном заголовке указывается используемый веб-браузер. Данный параметр никогда не меняется в течение сеанса, однако такие заголовки отправляются с каждым запросом;
- `Host`; данный заголовок используется для полного определения URL запроса. Заголовок будет повторяться для каждого запроса к одному и тому же хосту;
- `Accept`; этот заголовок определяет формат ожидаемого ответа (например, допустимые для браузера форматы изображений и т. д.). Поскольку форматы, поддерживаемые браузером, обычно меняются только при обновлениях, заголовок `Accept` меняется в зависимости от типа запроса (изображение, документ, шрифт и т. д.), однако для одного и того же события будет использоваться один и тот же запрос;
- `Accept-Encoding`; этот заголовок определяет форматы сжатия (обычно это алгоритмы `gzip`, `deflate`, а также `br` для браузеров, поддерживающих формат сжатия brotli). Как и заголовок `Accept`, в течение сеанса он остается неизменным.

Вышеописанные заголовки ответов могут дублироваться, а также напрасно занимать большой объем. Некоторые специализированные за-

головки, например заголовки *политики безопасности контента* (Content Security Policy), могут быть весьма большими. Особенно серьезной проблемой такие заголовки становятся в небольших запросах, в которых они составляют пропорционально большую часть всего объема.

HTTP/1 позволяет сжимать тело HTTP (посредством заголовка `Accept-Encoding`, как упоминалось выше), но не заголовки. HTTP/2 позволяет сжимать заголовки, но, как мы обсудим позже в главе 8, вместо сжатия тел он вводит новый метод и разрешает сжатие перекрестных запросов, предотвращая тем самым возникновение проблем безопасности при использовании алгоритмов сжатия тел запросов HTTP.

4.1.5 Server push

Еще одно важное различие между HTTP/1 и HTTP/2 заключается в том, что HTTP/2 добавляет концепцию *push-загрузки*, которая позволяет серверу отвечать на запрос более чем одним ответом. При использовании HTTP/1 при получении запрошенной домашней страницы браузер должен сначала обработать ее, а затем запросить другие ресурсы (такие как CSS и JavaScript) и только потом начать отображение страницы. С помощью концепции push-загрузки, введенной в HTTP/2, вышеупомянутые ресурсы могут быть отправлены вместе с первым ответом. Кроме того, они должны быть доступны для использования браузером при необходимости.

Push-загрузка в протоколе HTTP/2 – это новая полезная концепция HTTP, однако при неправильной реализации она может привести к потере пропускной способности. Так происходит, когда ресурсы не нужны браузеру, но все равно отправляются сервером. Это особенно невыгодно, если они дублируют предыдущие запросы и находятся в кеше браузера. Правильное внедрение данной концепции является основным аспектом максимально эффективного использования этой функции. Именно поэтому технологии push-загрузки HTTP/2 посвящена отдельная глава нашей книги (глава 5).

4.2 Как устанавливается HTTP/2-соединение

HTTP/2 сильно отличается от HTTP/1 на уровне соединения, поэтому для правильного использования клиентский браузер и сервер должны уметь обрабатывать и отправлять HTTP/2-сообщения. Ввиду того, что в этом процессе задействованы две независимые стороны, должен быть процесс, в ходе которого каждая сторона должна иметь возможность сказать, что желает и может использовать HTTP/2.

Такая возможность появилась благодаря переходу на HTTPS, который был осуществлен на основе новой схемы URL (`https://`). HTTPS-соединения обслуживаются по умолчанию через другой порт (если для HTTP это был порт 80, то для HTTPS 443). Переход позволил четко разделить протоколы и явно указать, какой из них используется для связи.

Однако у перехода на новую схему или новый порт (или и то и другое) есть несколько недостатков, например:

- пока переход не станет практически всеобщим, по умолчанию необходимо оставить схему http:// (или https://, если, как надеются многие, она когда-либо станет схемой по умолчанию). Следовательно, добавление новой схемы, как подразумевает HTTPS, потребовало бы перенаправления для использования HTTP/2, что неизбежно привело бы к снижению скорости соединения (как раз к тому, с чем HTTP/2 призван бороться);
- сайтам необходимо поменять ссылки в соответствии с новой схемой. Внутренние ссылки можно легко заменить относительными (например, /images/image.png, а не https://example.com/images/image.png), однако внешние ссылки должны включать в себя полный URL, в том числе и схему. Многим сайтам было сложно перейти на HTTPS ввиду того, что им приходилось менять каждый URL;
- возникают проблемы совместимости с существующей сетевой инфраструктурой (например, брандмауэры блокируют любые новые нестандартные порты).

Именно по вышеперечисленным причинам, а также с целью облегчить переход на HTTP/2 для любых серверов, в HTTP/2 (и SPDY, лежащем в его основе) разработчики решили отказаться от новой схемы, но предусмотрели альтернативные методы создания HTTP/2-соединения. В спецификации HTTP/2[1] описаны три способа создания соединения (кроме того, позже был добавлен еще один способ, упомянутый в разделе 4.2.4):

- использование HTTPS-соединений;
- использование HTTP-заголовка `Upgrade`;
- использование заранее известного протокола.

В теории HTTP/2 доступен при использовании как незашифрованных соединений (HTTP), где он указывается как h2c, так и зашифрованных (HTTPS) под именем h2. На практике же все веб-браузеры поддерживают HTTP/2 только при HTTPS-соединениях (h2). Вариант подключения через HTTP применяется для согласования HTTP/2 браузерами. Обмен данными между серверами HTTP/2 может осуществляться по незашифрованному протоколу HTTP (h2c) или HTTPS (h2), поэтому он может использовать любой метод в зависимости от того, какая схема используется.

4.2.1 Использование HTTPS-рукопожатия

Одним из важных этапов настройки HTTPS-соединения является этап *согласования протокола*. Перед установкой соединения и началом обмена HTTP-сообщениями необходимо согласовать протокол SSL/TLS, шифр и другие необходимые параметры. Данный этап является достаточно гибким, так как он позволяет вводить и использовать новые протоколы

[1] https://tools.ietf.org/html/rfc7540#section-3.

и шифры, однако только в случае согласия обеих сторон. Частью HTTPS-рукопожатия может быть и соглашение о поддержке HTTP/2. В таком случае любые перенаправления для обновления протокола, которые должны были быть выполнены при установке соединения, сохраняются до момента отправки первого сообщения.

HTTPS-рукопожатие

И в HTTP/1, и в HTTP/2 HTTPS-соединения представляют собой стандартные HTTP-соединения, зашифрованные с помощью SSL/TLS. Для того чтобы разобраться в различиях между всеми вышеперечисленными названиями, обратитесь к врезке «SSL, TLS, HTTPS и HTTP» в главе 1.

Шифрование с открытым ключом называют *ассиметричным шифрованием*, поскольку для шифрования и дешифрования сообщений здесь используются разные ключи. Данный тип шифрования позволяет обеспечить безопасную связь с сервером, к которому вы подключаетесь впервые, но работает медленно, поэтому используется для согласования симметричного ключа, впоследствии применяемого для шифрования остальной части соединения. Это согласование происходит в самом начале, во время TLS-рукопожатия. В TLSv1.2, который в данный момент является основной версией, для установки зашифрованного соединения также используется рукопожатие (см. рис. 4.4). В версии TLSv1.3, для которой недавно был выпущен стандарт, рукопожатие немного изменилось, о чем мы поговорим в главе 9.

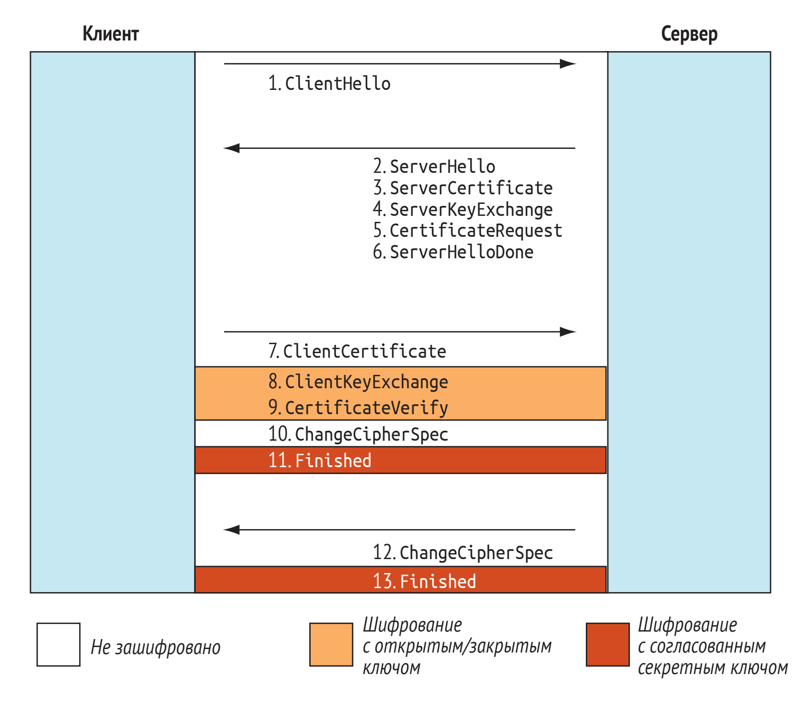


Рис. 4.4 HTTPS-рукопожатие

Рукопожатие включает в себя четыре набора сообщений:

- клиент отправляет сообщение `ClientHello`, где подробно описываются его возможности шифрования. Поскольку метод шифрования еще не согласован, данное сообщение отправляется в незашифрованном виде;
- в ответ сервер отправляет аналогичное сообщение – `ServerHello`, где указывается протокол HTTPS (например, TLSv1.2), который он выбрал на основе отправленных клиентом данных. Также он выбирает и отправляет клиенту шифр, который будет использоваться для данного соединения (например, `ECDHE-RSA-AES128-GCM-SHA256`). Затем он отправляет сертификат HTTPS (`ServerCertificate`). Детали секретного ключа зависят от выбранного шифра (`ServerKeyExchange`) и от необходимости наличия клиентского сертификата HTTPS (`CertificateRequest`, не требуется для большинства веб-сайтов). После того как все предыдущие пункты выполнены, сервер сообщает, что все готово (`ServerHelloDone`);
- клиент проверяет сертификат сервера и при следующем запросе отправляет свой сертификат (`ClientCertificate`, не требуется для большинства сайтов). Затем он отправляет сведения о своем секретном ключе (`ClientKeyExchange`). Эти данные отправляются в зашифрованном виде с помощью открытого ключа согласно сертификату сервера, поэтому дешифровать их может только сервер (с помощью секретного ключа). Если сертификат клиента все же потребовался, отправляется сообщение `CertificateVerify`, подписанное личным ключом с целью подтвердить право собственности на сертификат клиента. Клиент использует данные `ServerKeyExchange` и `ClientKeyExchange` для определения зашифрованного симметричного ключа и отправляет сообщение `ChangeCipherSpec`, чтобы проинформировать клиента о начале шифрования, затем отправляется зашифрованное сообщение `Finished`;
- сервер переключается на зашифрованное соединение (`ChangeCipherSpec`) и отправляет зашифрованное сообщение `Finished`.

Криптография с открытым ключом, помимо согласования симметричного ключа шифрования, используется также и для подтверждения личности, так как данный аспект является одной из основ безопасности. Идентификация выполняется успешно, если сообщения подписываются секретным личным ключом сервера, который можно разблокировать с помощью открытого ключа в сертификате. Каждый сертификат SSL/TLS также криптографически подписан доверенным центром сертификации. Аналогичный процесс работает и с клиентскими сертификатами, но только в обратном порядке. Кроме того, домен сервера является частью подписанного сертификата SSL/TLS. Если домен сервера неверен (например, www.amaz0n.com, а не www.amazon.com), возможно, вы установили соединение совсем не с той стороной, с которой предполагали. Как уже упоминалось в главе 1, эта ситуация вызывает большую путаницу. Значок зеленого замка не гарантирует безопасность сайта, он гарантирует только то, что обмен данными с ним надежно зашифрован.

После того как все вышеописанные шаги были выполнены, сеанс HTTPS полностью настроен, и все дальнейшие коммуникации защищены согласованным ключом (ключами). Таким образом, перед отправкой первого запроса происходит как минимум два дополнительных цикла приема и передачи данных. HTTPS всегда считался довольно медленным, поскольку даже несмотря на то, что благодаря техническому прогрессу шифрование и дешифрование отнимают все меньше времени, задержка при использовании данного протокола все же заметна, как вы можете видеть на каскадных диаграммах в главе 2. После настройки сеанса остальным HTTP-сообщениям уже не требуется согласование. Точно так же дальнейшие соединения (будь то дополнительные параллельные или повторные соединения) могут *возобновить* сеанс TLS, если они используют ключи повторно.

Проблему с задержкой нельзя решить никак, кроме попытки ограничить создание новых соединений (как это делает HTTP/2). Большинство людей сходится во мнении, что HTTPS все же имеет больше преимуществ, чем недостатков. На момент написания этой книги версия TLSv1.3[1] уже полностью доработана. Данная версия работает намного эффективнее, поскольку позволяет сократить количество циклов приема и передачи до одного (или даже вообще отказаться от них при переходе от предыдущего согласования). Однако для ее принятия потребуется некоторое время, и вам все же придется проходить хотя бы один дополнительный цикл.

Согласование протокола уровня приложений

ALPN[2] расширил сообщение `ClientHello`. В этом расширении клиенты могут объявлять о том, что они поддерживают протокол уровня приложений («Эй, я поддерживаю h2 и http/1. Если хотите, используйте любой из них».). Кроме того, изменилось и сообщение `ServerHello`, благодаря чему серверы могут подтверждать, какой протокол приложения они будут использовать после согласования HTTPS («Хорошо, давайте использовать h2».). Весь вышеописанный процесс изображен на рис. 4.5.

ALPN – довольно простой протокол. Он может использоваться для согласования HTTP/2 для уже существующих HTTPS-сообщений без добавления каких-либо дополнительных циклов передачи, перенаправлений или других задержек, обычно возникающих при обновлении. Единственная проблема ALPN – это то, что он относительно новый, и не все пользователи поддерживают его. Например, его не поддерживают многие серверы, где используются более старые версии TLS-библиотек (см. главу 3). Если не поддерживается ALPN, сервер обычно предполагает, что клиент не поддерживает HTTP/2 и переходит на HTTP/1.1.

Помимо HTTP/2, ALPN может использоваться и для других протоколов. На момент написания нашей книги он поддерживает работу с HTTP/2 и SPDY, лежащим в его основе. Однако были зарегистрированы и дру-

---

1 https://tools.ietf.org/html/draft-ietf-tls-tls13.

2 https://tools.ietf.org/html/rfc7301.

гие применения ALPN, включая три начальных версии HTTP: HTTP/0.9, HTTP/1.0 и HTTP/1.1.6[1]. Разработка ALPN была завершена в июле 2014 года, еще до появления HTTP/2, а RFC для ALPN[2] содержит информацию только для HTTP/1.1 и SPDY. Расширение ALPN для HTTP/2 (h2) было зарегистрировано позже в составе спецификации HTTP/2[3].

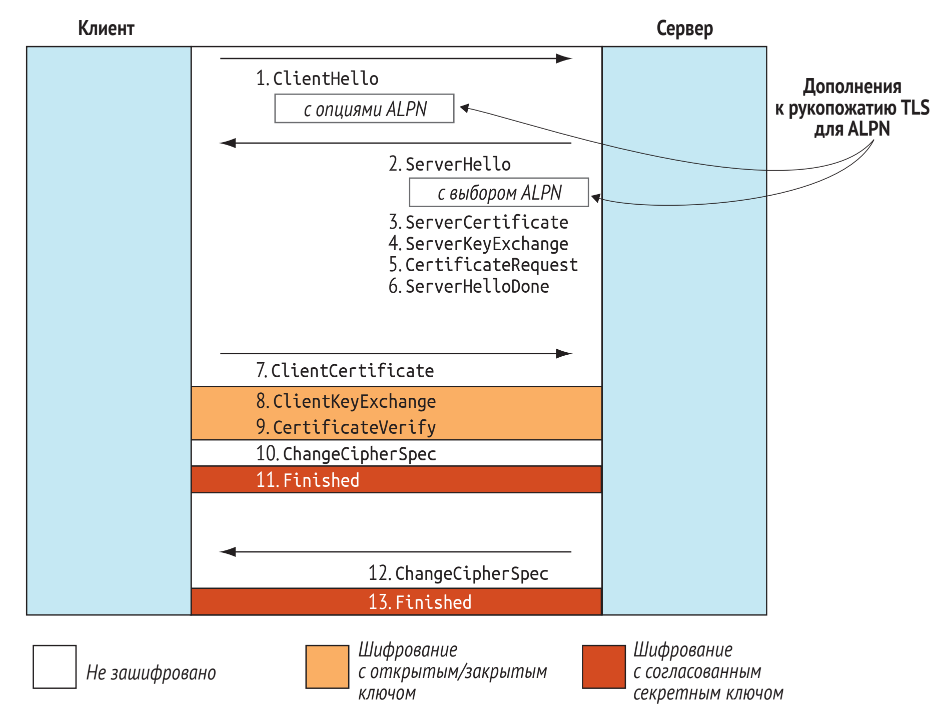


Рис. 4.5 HTTPS-рукопожатие с ALPN

Next Protocol Negotiation

NPN (Next Protocol Negotiation), предшественник ALPN, работает по аналогичному принципу. Несмотря на то что его использовало множество браузеров и серверов, он так и не был оформлен в качестве официального Internet-стандарта (хотя был разработан даже проект спецификации)[4]. Стандарт ALPN в свою очередь был официально оформлен. Он в значительной степени основан на NPN аналогично тому, как HTTP/2 является стандартизированной версией SPDY.

Основное отличие состоит в том, что при использовании NPN выбор, какой протокол использовать, остается за клиентом, а при использова-

---

1 https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml#alpn-protocol-ids.

2 https://tools.ietf.org/html/rfc7301.

3 https://tools.ietf.org/html/rfc7540#section-11.1.

4 https://tools.ietf.org/html/draft-agl-tls-nextprotoneg-04.

нии ALPN это делает сервер. В первом случае в сообщении `ClientHello` клиент указывает, что хотел бы использовать NPN. Сообщение `ServerHello` содержит перечень всех поддерживаемых сервером протоколов NPN, а затем, после установки зашифрованного соединения, клиент выбирает протокол NPN (например, h2) и отправляет сообщение, оповещающее сервер об этом выборе. Рисунок 4.6 иллюстрирует этот процесс; обратите внимание на три элемента, выделенных на этапах 1, 2 и 11.

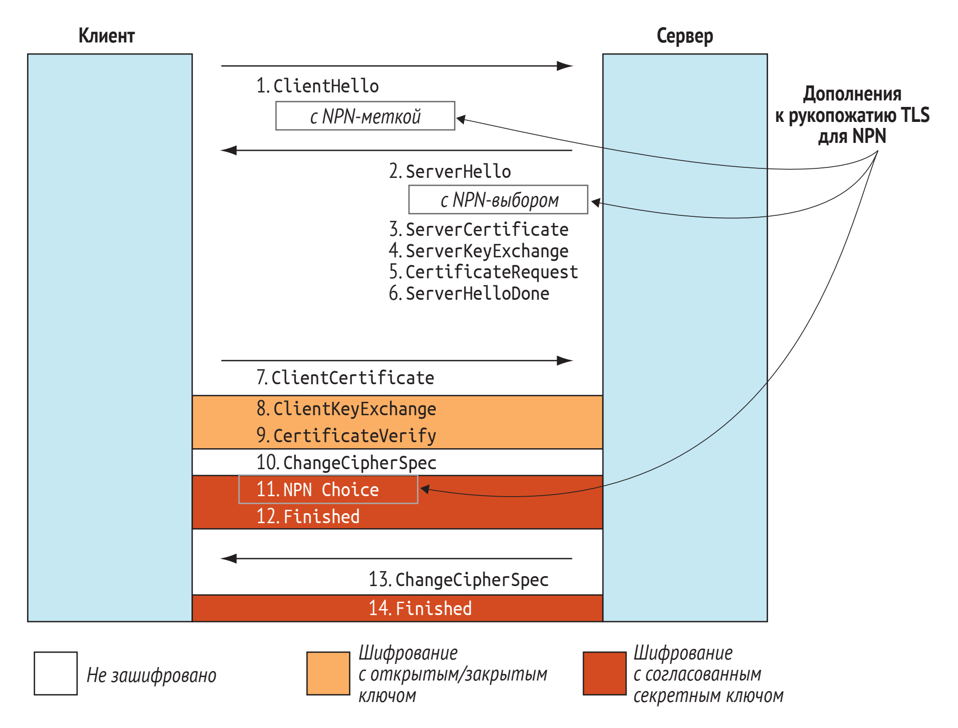


Рис. 4.6 HTTPS-рукопожатие с NPN

NPN включает в себя три этапа, тогда как ALPN – два. Однако они оба повторно используют уже существующие шаги установки HTTPS-соединения и не добавляют новых циклов приема и передачи (хотя стоит отметить, что NPN добавляет одно сообщение для подтверждения выбранного протокола). Кроме того, при использовании NPN информация о выбранном протоколе приложения будет зашифрована (этап 11 на рис. 4.6, тогда как ALPN отправляет ее в незашифрованном сообщении `ServerHello`. Ввиду того, что в NPN эти данные отправляются в незашифрованном виде в сообщении `ServerHello` и поскольку некоторым сетевым решениям необходимо владеть информацией о применимом протоколе, рабочая группа TLS решила внести изменения в этот процесс в ALPN, где выбор протокола приложения (и других параметров HTTPS) остается за сервером.

По сравнению с ALPN, NPN считается устаревшим, и, как было сказано в главе 3, многие браузеры перестали поддерживать его для соединений

HTTP/2; некоторые веб-серверы (например, Apache) вообще никогда не поддерживали его. Ожидается, что со временем и другие реализации прекратят его использование. В спецификации HTTP/2 сказано, что следует использовать ALPN[1], а NPN не упоминается вовсе, поэтому с технической точки зрения реализации, продолжающие использовать NPN, не соответствуют спецификации.

Повсеместное прекращение поддержки NPN браузерами становится серьезной проблемой для серверов, которые еще не перешли на ALPN. Ввиду того, что NPN старше, его поддерживало большинство серверов (или по крайней мере их TLS-библиотеки). Более новые версии поддерживают ALPN, однако старые версии, коих на момент написания нашей книги большинство (например, OpenSSL 1.0.1), поддерживают только старое расширение. К слову, такая ситуация является одной из причин, по которой невозможно использовать HTTP/2 даже при должной настройке сервера (см. главу 3).

Пример HTTPS-рукопожатия с ALPN

Для просмотра HTTPS-рукопожатия вы можете использовать несколько инструментов. Самым простым и доступным из них является curl[2], однако, возможно, что ваша версия не поддерживает расширение ALPN. Впрочем, те читатели, которые пользуются Git Bash, могут не переживать по этому поводу, ведь он включает в себя версию с поддержкой ALPN. Пример ниже иллюстрирует, что происходит, когда вы подключаетесь к Facebook с помощью HTTP/2 и инструмента curl. Жирным шрифтом выделены части кода, относящиеся к ALPN и HTTP/2:

```
$ curl -vso /dev/null --http2 https://www.facebook.com
* Rebuilt URL to: https://www.facebook.com/
* Trying 31.13.76.68...
* TCP_NODELAY set
* Connected to www.facebook.com (31.13.76.68) port 443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
* CAfile: /etc/pki/tls/certs/ca-bundle.crt CApath: none } [5 bytes data]
* TLSv1.2 (OUT), TLS handshake, Client hello (1): } [214 bytes data]
* TLSv1.2 (IN), TLS handshake, Server hello (2): { [102 bytes data]
* TLSv1.2 (IN), TLS handshake, Certificate (11): { [3242 bytes data]
* TLSv1.2 (IN), TLS handshake, Server key exchange (12): { [148 bytes data]
* TLSv1.2 (IN), TLS handshake, Server finished (14): { [4 bytes data]
* TLSv1.2 (OUT), TLS handshake, Client key exchange (16): } [70 bytes data]
* TLSv1.2 (OUT), TLS change cipher, Client hello (1): } [1 bytes data]
* TLSv1.2 (OUT), TLS handshake, Finished (20): } [16 bytes data]
* TLSv1.2 (IN), TLS handshake, Finished (20): { [16 bytes data]
* SSL connection using TLSv1.2 / ECDHE-ECDSA-AES128-GCM-SHA256
* ALPN, server accepted to use h2
* Server certificate:
```

1 https://tools.ietf.org/html/rfc7540#section-3.3.

2 https://curl.haxx.se/.

```
* subject: C=US; ST=California; L=Menlo Park; O=Facebook, Inc.;
CN=*.facebook.com
* start date: Dec 9 00:00:00 2016 GMT
* expire date: Jan 25 12:00:00 2018 GMT
* subjectAltName: host "www.facebook.com" matched cert's "
*.facebook.com" * issuer: C=US; O=DigiCert Inc; OU=www.digicert.com; CN=DigiCert
SHA2
High
Assurance Server CA
* SSL certificate verify ok.
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
```

В данном примере клиент заявляет, что для поддержки HTTP/2 (h2) и HTTP/1.1 (http/1.1) он будет использовать ALPN. Затем он проходит многочисленные этапы подтверждения (показаны не все, но этого достаточно для создания общего представления), и наконец устанавливается соединение с `TLSv1.2`, шифр `ECDHE-ECDSA-AES128-GCM-SHA256` и `h2` ALPN. После этого отображается сертификат сервера, и curl переключает сеанс на HTTP/2. Curl может помочь вам проверить, поддерживает ли сервер ALPN (конечно, при условии, что ваша версия curl поддерживает ALPN). Если вам интересно, вы можете протестировать NPN с помощью опции `--no-alpn`. Этот тест, однако, не покажет вам столь большой объем информации и исключит все строки ALPN, которые можно увидеть в предыдущем примере, без замены каких-либо эквивалентов NPN, хотя последние две строки идентичны в обоих примерах:

```
$ curl -vso /dev/null --http2 https://www.facebook.com --no-alpn
...
* SSL certificate verify ok.
* Using HTTP2, server supports multi-use
* Connection state changed (HTTP/2 confirmed)
```

4.2.2 HTTP-заголовок `Upgrade`

Клиент может запросить обновление существующего HTTP/1.1-соединения до HTTP/2, отправив HTTP-заголовок `Upgrade`. Данный заголовок следует использовать только для незашифрованных HTTP-соединений (h2c). При работе с HTTPS-соединениями HTTP/2 (h2) не следует согласовывать HTTP/2 с помощью этого заголовка – согласование HTTPS должно происходить с помощью ALPN. Как уже было отмечено, веб-браузеры поддерживают HTTP/2 только через зашифрованные соединения, следовательно, для них этот метод не нужен. Для тех, кто работает с внешними приложениями, например через API, может быть интересно узнать более подробную информацию о том, как работает данный метод.

Если заголовок `Upgrade` отправляет клиент, то он полностью зависит от отправителя. Заголовок может отправляться с каждым запросом, только с начальным запросом или только в тех случаях, когда сервер объявляет о поддержке HTTP/2 с помощью заголовка `Upgrade` в HTTP-ответе. Примеры ниже показывают, как работает данный заголовок.

Пример 1: неудачный запрос с заголовком Upgrade

Здесь запрос HTTP/1.1 выполняется с использованием данного заголовка, поскольку клиент поддерживает HTTP/2 и хочет использовать именно его:

```
GET / HTTP/1.1
Host: www.example.com
Upgrade: h2c
HTTP2-Settings: <will be discussed later>
```

Такой запрос должен включать в себя заголовок `HTTP-Settings`, который представляет собой настройки HTTP/2 в кодировке base-64, о которых мы поговорим позже.

Сервер, не поддерживающий HTTP/2, отвечает на запрос сообщением HTTP/1.1, проигнорировав заголовок `Upgrade`:

```
HTTP/1.1 200 OK
Date: Sun, 25 Jun 2017 13:30:24 GMT
Connection: Keep-Alive
Content-Type: text/html
Server: Apache

<!doctype html>
<html>
<head>
…и т. д.
```

Пример 2: успешный запрос с заголовком Upgrade

Сервер, поддерживающий HTTP/2, вместо того чтобы игнорировать запрос на обновление и отправлять обратно ответ `HTTP/1.1 200`, присылает в ответ `HTTP/1.1 101`, что свидетельствует о переходе на использование нового протокола:

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c
```

Затем сервер немедленно переключается на HTTP/2 и отправляет сначала фрейм `SETTINGS` (см. раздел 4.3.3), а после – ответ на исходное сообщение уже в формате HTTP/2.

Пример 3: заголовок Upgrade отправляет сервер

Итак, клиент выполняет запрос HTTP/1.1, и предполагает, что сервер не поддерживает HTTP/2, поэтому в запросе нет заголовка `Upgrade`:

```
GET / HTTP/1.1
Host: www.example.com
```

Однако, если сервер поддерживает HTTP/2, он отправляет ответ с кодом `200` и включает в него заголовок `Upgrade`. Но этот заголовок будет являться *предложением*, а не запросом на обновление, поскольку все по-

добные запросы инициируются клиентом. Ниже приведен пример, где сервер сообщает о поддержке h2 (HTTP/2 через HTTPS) и h2c (HTTP/2 через HTTP):

```
HTTP/1.1 200 OK
Date: Sun, 25 Jun 2017 13:30:24 GMT
Connection: Keep-Alive
Content-Type: text/html
Server: Apache
Upgrade: h2c, h2

<!doctype html>
<html>
<head>
…и т. д.
```

Клиент в свою очередь может использовать эту информацию чтобы инициировать переход на HTTP/2. Для этого он отправляет заголовок `Upgrade` в следующем запросе:

```
GET /styles.css HTTP/1.1
Host: www.example.com
Upgrade: h2c
HTTP2-Settings: <will be discussed later>
```

На этот запрос сервер даст ответ `101` и обновит соединение, как это было описано ранее. Обратите внимание, что заголовок `Upgrade` и метод согласования нельзя использовать для h2-соединений – только для h2c-соединений. В данном примере сервер объявил как о h2, так и о h2c, но, если клиент хочет использовать h2, ему следует переключиться на HTTPS и использовать ALPN для согласования.

Проблемы с отправкой заголовка Upgrade

На момент написания этой книги все браузеры поддерживают HTTP/2 только с HTTPS. Именно поэтому заголовок `Upgrade`, скорее всего, никогда не будет использоваться браузерами, что может вызвать некоторые проблемы.

Рассмотрим этот сценарий. Предположим, что на одной стороне соединения находится браузер, поддерживающий HTTP/2, а на другой перед сервером приложений, поддерживающим HTTP/2, находится еще один сервер, который поддерживает только HTTP/1.1 (например, более старая версия Apache). В таком случае последний действует как обратный прокси и, вероятно, пропускает через себя все запросы клиента (веб-браузера) и конечного сервера (сервер приложений Tomcat), которые поддерживают HTTP/2. Сервер приложений может попытаться отправить заголовок `Upgrade`, чтобы перейти на более эффективную версию протокола. Промежуточный сервер может вслепую переслать этот заголовок, и клиент решит, что это действительно неплохая идея. Однако впоследствии окажется, что промежуточный сервер все-таки не поддерживает HTTP/2.

В похожем сценарии веб-сервер передает браузеру информацию в формате HTTP/2, а на сервер приложений проксирует запросы с помощью HTTP/1.1. Сервер приложений может отправить предложение о переходе на новую версию протокола, и, если это предложение будет перенаправлено к браузеру, может возникнуть путаница, поскольку браузер и так использует соединение h2.

Подобные проблемы довольно часто возникали на практике до того, как был развернут HTTP/2. Например, Safari отправлял ответ с кодом ошибки, когда видел h2-заголовок `Upgrade` в соединении HTTP/2 (см. главу 3).

На момент написания данной книги команда nginx исправила ошибку, при которой заголовок `Upgrade` передавался вслепую[1], например сервером Apache. Некоторые конфигурации (такие как `proxy_hide_header Upgrade`) позволяют скрыть данный заголовок, однако мало кто знает, как добавить его обратно во избежание проблем. Кроме того, некоторые клиенты или серверы могут неправильно реализовать заголовок `Upgrade`. Во время экспериментов с HTTP/2 была замечена проблема с отключением NodeJS, после того как Apache передавал заголовок `Upgrade`[2]. В новых версиях NodeJS данная проблема была исправлена, однако в старых, которые все еще используются, она присутствует.

К моменту публикации этой книги все описанные проблемы могут быть уже исправлены, однако не исключено и появление новых подобных проблем. Подводя итог всему вышесказанному, лучше, чтобы реализации сервера *не* предлагали обновить протокол (по крайней мере, по умолчанию). На мой взгляд, при таких сценариях этот подход привносит больше проблем, чем решает. Большинство версий серверов (и все веб-браузеры), скорее всего, будет использовать согласование HTTPS. Метод предположения протокола по умолчанию может использоваться для внутренних серверов, где HTTPS не поддерживается или не требуется. Одним из «нарушителей» является Apache, которому предложили прекратить использовать `Upgrade` по умолчанию[3]. Однако вы можете и самостоятельно отключить отправку данного заголовка в разделе конфигурации Apache `mod_headers`. Если вы используете Apache с поддержкой HTTP/2, мы настоятельно советуем сделать это (хотя это решение может вызвать проблемы с другими протоколами, которым необходимо использовать заголовок `Upgrade`, например WebSockets, поэтому его нельзя использовать в некоторых случаях):

```
Header unset Upgrade
```

4.2.3 Применение заранее известного протокола

Третий и последний способ перехода на HTTP/2, согласно спецификации, заключается в том, что клиент уже заранее знает, что сервер поддержи-

1 https://trac.nginx.org/nginx/ticket/915.
2 https://github.com/nodejs/node/issues/4334.
3 https://bz.apache.org/bugzilla/show_bug.cgi?id=59311.

вает HTTP/2. В этом случае можно сразу запустить HTTP/2-соединение, не прибегая к использованию запросов на обновление.

Клиент может получить предварительную информацию о том, что сервер поддерживает HTTP/2 несколькими разными способами. Например, если вы используете обратный прокси для разгрузки HTTPS, возможно, вы предпочтете сразу передавать сообщения HTTP/2 своим внутренним серверам через HTTP (h2c), так как вы знаете, что они поддерживают HTTP/2. Кроме того, вы можете получить предварительную информацию из альтернативного источника благодаря заголовку `Alt-Svc` или фрейму `ALTSVC` (см. раздел 4.2.4).

Данный вариант является весьма рискованным, так как информация о том, что сервер использует HTTP/2, может быть не совсем точной. Клиенты, прибегнувшие к нему, должны позаботиться о том, чтобы любые сообщения обрабатывались надлежащим образом, даже если предварительная информация окажется неверной. Ответ сервера на предварительное сообщение, касающееся HTTP/2 (о котором мы поговорим позже), имеет огромное значение при выборе такого способа использования. Данный метод следует использовать только в случае, если вам доступно управление и клиентом, и сервером.

4.2.4 Протокол HTTP Alternative Services

Четвертый способ, не прописанный в спецификации HTTP/2, заключается в использовании протокола HTTP Alternative Services (альтернативные службы HTTP)[1], который впоследствии был оформлен в отдельный стандарт уже после публикации HTTP/2. Он позволяет серверу с помощью HTTP/1.1 информировать клиента (через HTTP-заголовок `Alt-Svc`) о том, что запрошенный ресурс доступен в другом месте (например, на другом сервере или порту) с использованием другого протокола. Данный протокол можно использовать для запуска HTTP/2 в соответствии с предварительно полученной информацией.

HTTP Alternative Services предназначен не только для HTTP/1, но и для HTTP/2 (через новый фрейм `ALTSVC`, о котором мы поговорим позже) в случае, если клиент хочет переключиться на другое соединение (например, которое расположено ближе или меньше загружено). Этот стандарт относительно молод и еще не получил широкого распространения. Именно поэтому по-прежнему происходит запуск одного соединения с последующим переключением. Данный способ занимает намного больше времени, чем запуск HTTP/2 через ALPN или на основании заранее полученной информации. Кроме того, он открывает некоторые интересные возможности, которые выходят за рамки нашей книги, но, похоже, только одна сеть доставки контента намерена в полной мере использовать его[2].

---

1 https://tools.ietf.org/html/rfc7838.

2 https://blog.cloudflare.com/cloudflare-onion-service/.

4.2.5 Преамбула соединения HTTP/2

Первое сообщение, которое должно быть отправлено в соединении HTTP/2 (независимо от того, какой метод используется для установления этого соединения), – это *преамбула соединения*, или так называемая «волшебная» строка. Это первое сообщение клиента в новом соединении. Оно представляет собой последовательность из 24 октетов. В шестнадцатеричном представлении сообщение выглядит следующим образом:

```
0x505249202a20485454502f322e300d0a0d0a534d0d0a0d0a
```

В кодировку ASCII (American standard code for information interchange) это переводится так:

```
PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n
```

Это сообщение может показаться странным, однако оно не случайно похоже на сообщение в формате HTTP/1:

```
PRI * HTTP/2.0↲
↲
SM↲
↲
```

`PRI` здесь – это HTTP-метод (вместо `GET` или `POST`), `*` – ресурс, а `HTTP/2.0` – номер версии HTTP. Затем идут два символа возврата каретки (поскольку нет заголовков запроса), за которыми следует тело запроса `SM`.

Назначение этого внешне бессмысленного сообщения становится понятным, когда клиент передает сообщение в формате HTTP/2 серверу, который его не поддерживает. Такой сервер пытается обработать сообщение так же, как он обрабатывает любые другие HTTP-сообщения и терпит неудачу. Он не может распознать бессмысленный для него метод (`PRI`) и версию HTTP (`HTTP/2.0`), поэтому отклоняет сообщение. Обратите внимание, что преамбула – это единственная часть официальной спецификации, где название версии употребляется с дополнительным номером (HTTP/2.0), в то время как везде пишется просто HTTP/2 (мы обсуждали это ранее во врезке раздела 4.1 «HTTP/2.0 или HTTP/2?». Сервер, который на основе входящего сообщения делает вывод о том, что клиент использует HTTP/2, не отправит подобный вышеописанному ответ. В качестве своего первого сообщения он отправит фрейм `SETTINGS` (который может быть пустым).

Зачем нужны PRI и SM?

Согласно ранним спецификациям протокола в HTTP/2 использовались идентификаторы FOO и BAR[a] или BA[b] – известные имена-заглушки в программировании. Однако в версии 4 рабочей спецификации они были без каких-либо разъяснений заменены на PRI и SM[c].

По всей видимости, такое нововведение было сделано в ответ на тогдашние разоблачения Эдварда Сноудена (Edward Snowden)[d] касательно программы

для сбора интернет-трафика различных компаний под названием PRISM. Сторонники свободного интернета (некоторые из которых связаны с разработкой Internet-стандартов) были расстроены такими открытиями и посчитали, что было бы забавно начинать каждое соединение HTTP/2 с небольшого напоминания.

Изначально содержание данного сообщения не имело никакого значения, ведь предполагалось что оно не должно распознаваться как действительное. Было предложено несколько других вариантов идентификаторов, например STA RT, но в конечном итоге именно идентификаторы PRI SM вошли в окончательную спецификацию с пометкой «Предпочтения редакционного совета касательно магической строки[e]».

[a] https://tools.ietf.org/html/draft-ietf-httpbis-http2-02#section-3.2.

[b] https://tools.ietf.org/html/draft-ietf-httpbis-http2-03#section-3.2.

[c] https://tools.ietf.org/html/draft-ietf-httpbis-http2-04#section-3.5.

[d] https://blog.jgc.org/2015/11/the-secret-message-hidden-in-every.html.

[e] https://github.com/http2/http2-spec/commit/ac468f3fab9f7092a430eedfd69ee1fb2e23c944.

4.3 Фреймы HTTP/2

После установки HTTP/2-соединения вы можете отправлять сообщения. Как я уже говорил, сообщения состоят из фреймов данных, которые отправляются в потоках по одному мультиплексированному соединению. Фреймы – это реализация низкого уровня, которой многие веб-разработчики не занимаются, однако всегда полезно понимать детали таких технологий. Многие ошибки, представленные в главе 3, легче устранить, разобрав HTTP/2 на уровне фреймов, именно поэтому данная информация имеет как теоретическое, так и практическое значение. Рассмотрим работу с фреймами HTTP/2 на реальных примерах.

В данном разделе мы изучим различные типы фреймов, которые на первый взгляд могут показаться запутанными и даже пугающими. Читателям, однако, следует сосредоточиться именно на общей концепции фреймов HTTP/2, а знать все о каждом из них необязательно. Далее в этой книге вы найдете разделы, посвященные отдельным фреймам и их настройкам, но вам не нужно заучивать всю эту информацию чтобы понять оставшуюся часть книги или разобраться в общих вопросах работы HTTP/2 в реальном мире.

4.3.1 Просмотр фреймов HTTP/2

Просматривать фреймы HTTP/2 можно с помощью ряда инструментов, в числе которых net-export в Chrome, nghttp и Wireshark. Кроме того, для того чтобы отображать отдельные фреймы, ваш веб-сервер может вести более подробные логи на уровне фреймов, однако при потенциально

большом количестве пользователей этот способ не подойдет. Поэтому проще использовать перечисленные выше инструменты (если конечно вы не пытаетесь устранить уже известную вам проблему).

Chrome net-export

Страница net-export в Chrome позволяет просматривать HTTP-фреймы без установки дополнительного программного обеспечения. Раньше такая функция была доступна в инструменте net-internals, но в Chrome 71 по ряду причин она была перемещена в net-externals[1], и теперь для того, чтобы просмотреть фреймы, требуется немного больше усилий. Для доступа к net-export необходимо открыть браузер Chrome и набрать в адресной строке следующее:

```
chrome://net-export/
```

Затем нажмите на кнопку **Start Logging to Disk** (Начать запись на диск) и выберите место для сохранения файла. В другой вкладке откройте сайт, использующий HTTP/2 (например, https://www.facebook.com), и после окончания его загрузки нажмите кнопку **Stop Logging** (Остановить запись). На этом этапе вы можете использовать средство просмотра NetLog (https://netlog-viewer.appspot.com), которое позволит открыть и проверить созданный файл (примечание: этот инструмент просматривает файл локально и не загружает его на сервер). Далее слева выберите параметр HTTP/2, а затем сайт (например, www.facebook.com). В результате вы должны увидеть соответствующие исходные сообщения HTTP/2, как показано на рис. 4.7.

Chrome добавляет много своей служебной информации, и, кроме того, многие фреймы разбиваются на несколько строк. Следующие строки взяты из одного фрейма `SETTINGS`:

```
t=  1646 [st=    1]  HTTP2_SESSION_RECV_SETTINGS
t=  1647 [st=    2]  HTTP2_SESSION_RECV_SETTING
                     --> id = "1 (SETTINGS_HEADER_TABLE_SIZE)"
                     --> value = 4096
t=  1647 [st=    2]  HTTP2_SESSION_RECV_SETTING
                     --> id = "5 (SETTINGS_MAX_FRAME_SIZE)"
                     --> value = 16384
t=  1647 [st=    2]  HTTP2_SESSION_RECV_SETTING
                     --> id = "6 (SETTINGS_MAX_HEADER_LIST_SIZE)"
                     --> value = 131072
t=  1647 [st=    2]  HTTP2_SESSION_RECV_SETTING
                     --> id = "3 (SETTINGS_MAX_CONCURRENT_STREAMS)"
                     --> value = 100
t=  1647 [st=    2]  HTTP2_SESSION_RECV_SETTING
                     --> id = "4 (SETTINGS_INITIAL_WINDOW_SIZE)"
                     --> value = 65536
```

[1] https://docs.google.com/document/d/1Ll7T5cguj5m2DqkUTad5DWRCqtbQ3L1q9FRvTN5-Y28/.

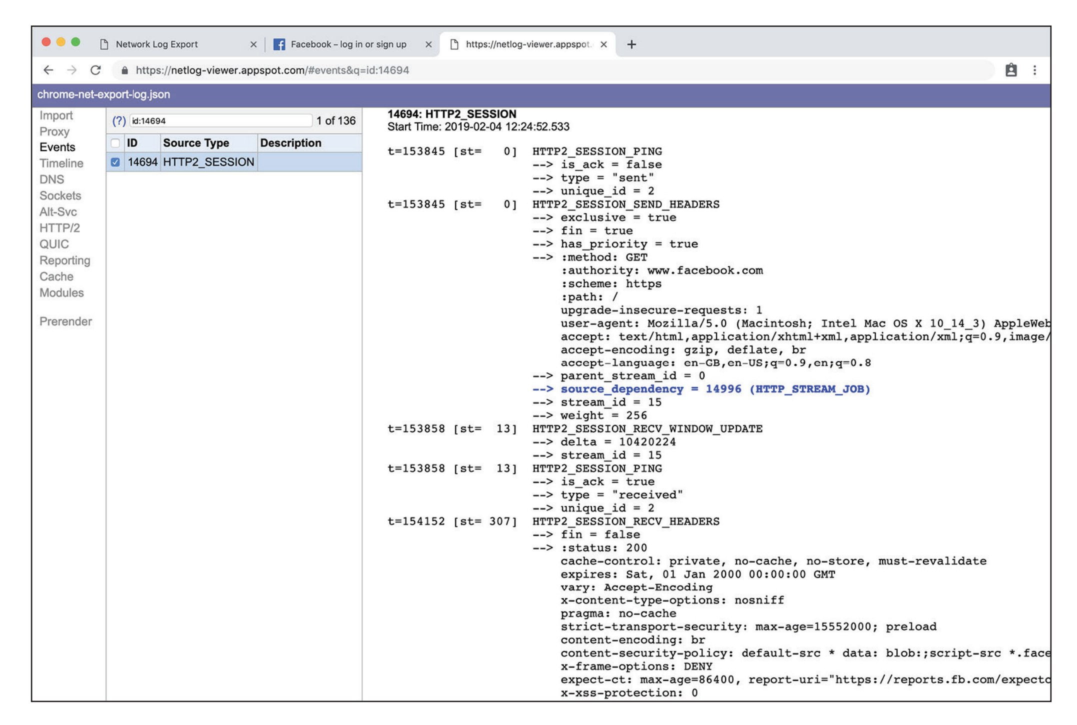


Рис. 4.7 Просмотр фреймов HTTP/2 в Chrome

При использовании данного инструмента читать отдельные фреймы может быть труднее, чем при помощи двух других инструментов, однако он отображает нам ту же информацию, что и они. С другой стороны, инструмент удобен тем, что вы можете получить такой уровень детализации в браузере без установки дополнительного программного обеспечения, а для форматирования результатов вы можете использовать дополнительные инструменты[1]. На момент написания данной книги мне не известны другие браузеры, обеспечивающие такой уровень детализации. Впрочем, Opera имеет аналогичные возможности ввиду того, что у нее база исходного кода такая же, как и у Chrome.

Использование nghttp

nghttp – это инструмент командной строки, разработанный на основе библиотеки nghttp2 на языке C. Веб-серверы и клиенты широко используют эту библиотеку для решения задач поддержки HTTP/2 на нижнем уровне. Если для вашего сервера установлена библиотека nghttp2 (они, например, требуются для Apache), возможно, у вас установлен данный инструмент. Вы можете пользоваться им для просмотра HTTP/2-сообщений так же, как и инструментом Chrome net-export. Взгляните на пример:

[1] https://github.com/rmurphey/chrome-http2-log-parser.

```
$ nghttp -v https://www.facebook.com
[  0.042] Connected
The negotiated protocol: h2
[  0.109] recv SETTINGS frame <length=30, flags=0x00, stream_id=0>
          (niv=5)
          [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
[  0.109] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=10420225)
[  0.109] send SETTINGS frame <length=12, flags=0x00, stream_id=0>
          (niv=2)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
…и т. д.
```

Использование Wireshark

Wireshark[1] позволяет отслеживать весь трафик, отправляемый и получаемый вашим компьютером. Данный инструмент может быть полезен при отладке системы на низком уровне, так как благодаря ему вы можете просматривать отправленные и полученные необработанные сообщения. К сожалению, пользоваться этим инструментом тоже довольно сложно.

Одна из сложностей заключается в том, что Wireshark не является клиентом. Он отслеживает трафик, который ваш браузер отправляет на сервер. Все браузеры используют HTTP/2 через HTTPS, поэтому, если вы не знаете, какие ключи шифрования и дешифрования SSL/TLS, вы не сможете прочитать трафик, что и является сутью HTTPS. Разработчики Chrome и Firefox приняли это во внимание, поэтому данные браузеры позволяют сохранять ключи HTTPS в отдельный файл, чтобы впоследствии вы могли использовать для отладки такие инструменты, как Wireshark. Очевидно, что при отладке необходимо выключить данный инструмент. Все, что вам нужно сделать, это указать браузеру (Chrome или Firefox) файл, в котором необходимо сохранить ключи. Вы можете сделать это, указав имя файла в переменной окружения `SSLKEYLOGFILE` или введя следующий код в командной строке запуска Chrome:

```
"C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" --ssl-key-log-
file=%USERPROFILE%\sslkey.log
```

> **ПРИМЕЧАНИЕ** Обязательно убедитесь, что вы используете правильные дефисы. Многие приложения, например Microsoft Office, автоматически заменяют короткий дефис (-) на короткое (–) или длинное тире (—). Запомните, что это три отдельных символа, которые командная строка может не распознать как указатель на аргументы. В результате получится пустой файл без ключей SSL, что может вызвать серьезные недоразумения.

---

[1] https://www.wireshark.org/.

Для macOS необходимо указать файл в переменной среды SSLKEYLOGFILE:

```
$ export SSLKEYLOGFILE=~/sslkey.log
$ /Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome
```

или упомянуть его как аргумент командной строки:

```
$ /Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome--ssl-key-
log-file=/Users/barry/sslkey.log
```

Затем вам необходимо запустить Wireshark и указать расположение файла. Для этого выберите **Edit** > **Preferences** > **Protocols** > **SSL** и укажите имя файла в поле **(Pre)-Master-Secret**, как показано на рис. 4.8.

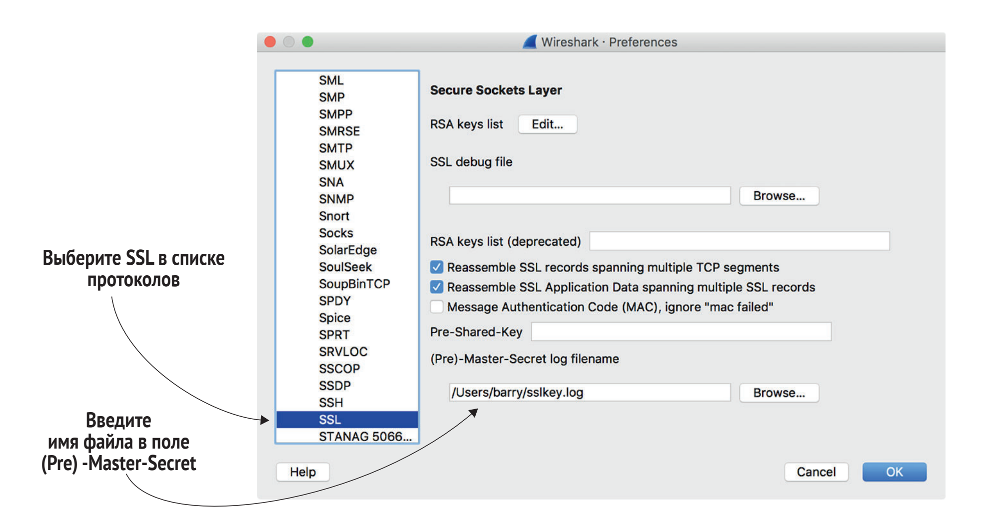


Рис. 4.8 Настройка файла секретного ключа HTTPS с помощью Wireshark

На данном этапе вам необходим доступ ко всем данным HTTPS, которые использует Chrome. Следовательно, если вы перейдете на https://www.facebook.com и проведете фильтрацию трафика в Wireshark по http2, вы сможете увидеть все сообщения, включая преамбулу соединения, о которой мы говорили в разделе 4.1.5 (доступно в Wireshark), что представлено на рис. 4.9.

На рис. 4.9 показано множество элементов и процессов. Если вы не пользовались Wireshark, это количество может показаться пугающим. В следующем списке описаны сегменты, пронумерованные на рисунке от 1 до 5.

1 В сегменте 1 можно задавать различные параметры фильтра. На рисунке показано, что отфильтрованы сообщения http2. Если у вас открыто много HTTP/2-соединений, вы можете конкретизировать фильтр, например фильтровать по IP-адресу сервера, к которому вы подключены. Следующий фильтр показывает только сообщения HTTP/2, отправленные на IP-адрес 31.13.90.2 и полученные с него

(сервер Facebook; найти его адрес можно в инструментах разработчика вашего браузера):

```
http2 && (ip.dst==31.13.90.2 || ip.src==31.13.90.2)
```

2 В следующем сегменте представлен список сообщений, соответствующих вашему фильтру. Для того чтобы получить подробную информацию, необходимо кликнуть на них.

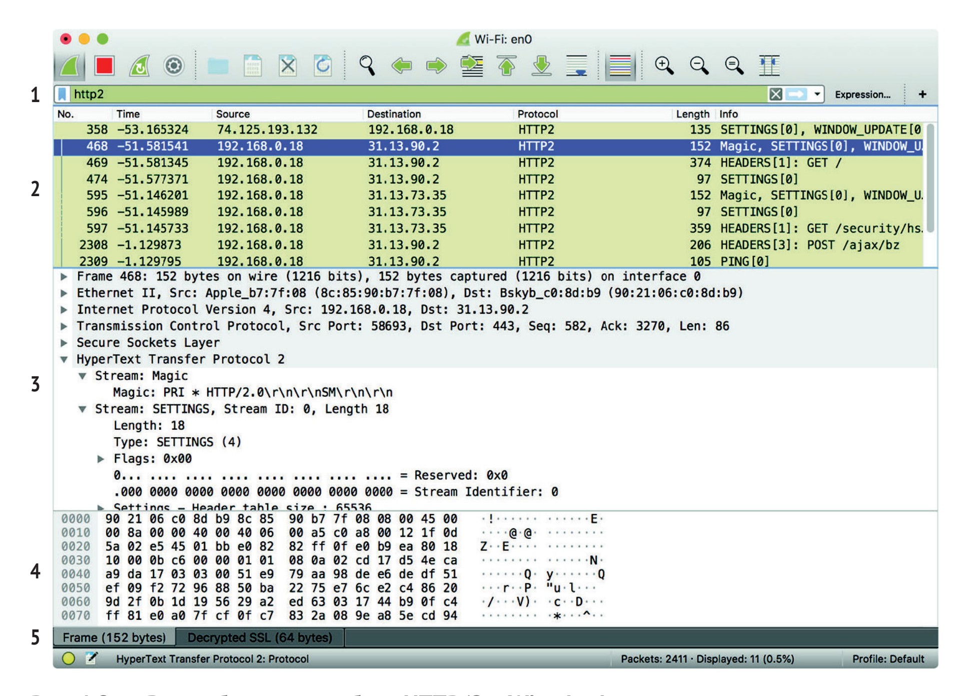


Рис. 4.9 «Волшебная» преамбула HTTP/2 в Wireshark

3 Далее представлена полная информация о сообщении. Если Wireshark распознает протокол (в большинстве случаев), он отобразит каждый протокол, к которому относится сообщение, в удобном для чтения формате.

Приведенный выше пример начинается с собственного базового фрейма (не путать с фреймами HTTP/2). Базовые фреймы отправляются как сообщения в сети Ethernet, которые обрабатываются IPv4, отправляются через TCP, затем через SSL/TLS и наконец преобразовываются в сообщения HTTP/2. Wireshark позволяет просматривать сообщения на любом из этих уровней. На предоставленном скриншоте развернуты сегменты HTTP/2 и «волшебной» преамбулы соединения. Если, например, вас больше интересует просмотр Ethernet, IP, TCP или уровень SSL/TLS, вы можете развернуть соответствующие сегменты аналогичным образом.

4 В предпоследнем сегменте показаны необработанные данные, обычно отображаемые в шестнадцатеричном формате и формате ASCII.

5 Вкладки в последнем сегменте помогут вам решить, как отображать необработанные данные. Наверняка формат распакованного заголовка (или, как для не имеющего сжатых заголовков «волшебного» сообщения, расшифрованный SSL) заинтересует вас больше, чем формат необработанного фрейма.

Кроме того, с помощью Wireshark вы можете просматривать сообщения согласования HTTPS (включая запросы расширения ALPN в сообщении `ClientHello` и ответы в сообщениях `ServerHello`). На рис. 4.10 стрелки указывают, что клиент выбирает сначала h2, а затем http/1.1.

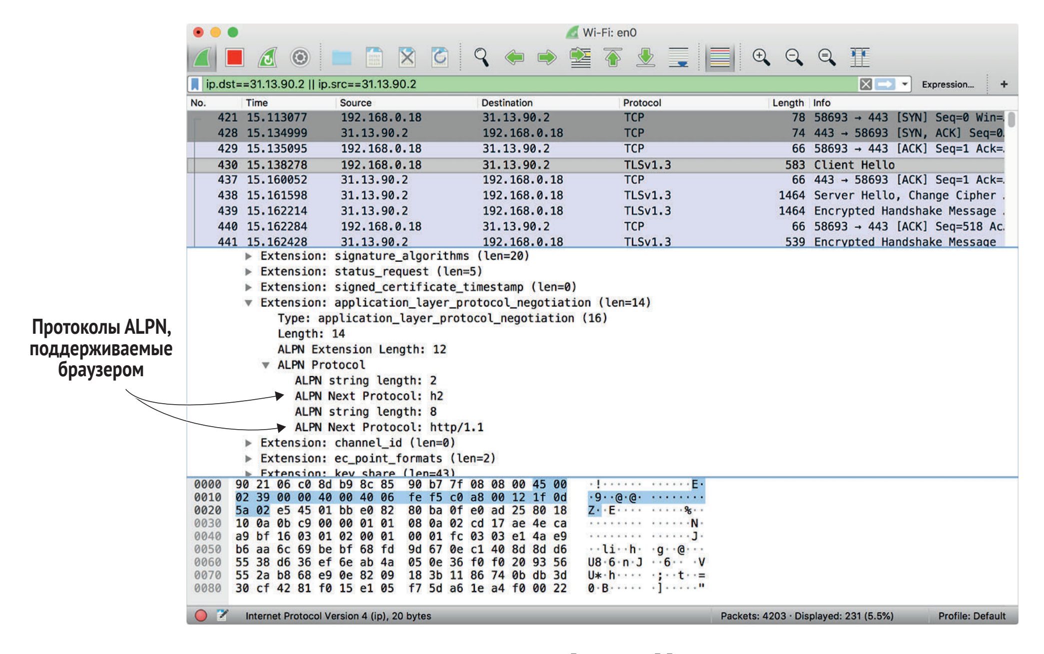


Рис. 4.10. Расширения ALPN как часть сообщения `ClientHello` в Wireshark

Проблемы с дешифровкой трафика Wireshark?

К сожалению, использование Wireshark для расшифровки HTTPS-трафика может быть, мягко говоря, немного нестабильным, поэтому вам, возможно, придется предпринять несколько попыток.

Одна из причин заключается в том, что данный процесс работает только для новых HTTPS-сеансов с полностью выполненным TLS-рукопожатием. Проблема состоит в том, что при последующих соединениях сайт может использовать настройки шифрования повторно, ввиду чего рукопожатие выполняется лишь частично, а этого недостаточно для расшифровки трафика с Wireshark. Чтобы узнать, используется ли возобновление сеанса HTTPS, установите фильтр на `ssl`, а не на `http2`. Посмотрите на первое сообщение `ClientHello` и проверьте, отличается ли значение идентификатора сеанса, а также проверьте, не являются ли значения `Session ID` и `SessionTicket` TLS не-

нулевыми. В таком случае текущий сеанс является возобновленным, и Wireshark не сможет расшифровать сообщения (если он не был запущен сразу после того, как установлен исходный сеанс).

Стоит также отметить, что ни один из браузеров не дает нам надежного способа удалить ключи SSL/TLS / удостоверения предыдущего сеанса, что могло бы обеспечить установку нового полного рукопожатия. Разработчики Chrome [a] и Firefox [b] получали множество просьб добавить такую функцию, однако реализация этой заявки была отложена на неопределенный срок.

Пользователи отмечают, что заставить Wireshark работать на macOS на порядок сложнее. Например, последняя версия Firefox отказалась от функции постоянного ведения лога ключей SSL.

Мы рекомендуем вам убедиться, что у вас установлена последняя версия Wireshark. Вы, конечно, можете возвращаться к сеансам по их истечению, однако такой способ малоэффективен. В качестве альтернативы вы можете попробовать альтернативный браузер, в котором предыдущий сеанс не сохраняется.

[a] https://bugs.chromium.org/p/chromium/issues/detail?id=90454.

[b] https://bugzilla.mozilla.org/show_bug.cgi?id=285440.

Какой инструмент выбрать?

Пользуйтесь тем инструментом, который наиболее удобен именно для вас, или, если хотите, каким-либо альтернативным инструментом. Wireshark нет равных, так как он предоставляет наиболее подробную информацию структуры и формата сообщений. Однако во многих случаях такой уровень детализации ни к чему. Кроме того, настроить данный инструмент довольно сложно. Если вы не знакомы с Wireshark, возможно, лучше использовать один из двух других инструментов.

В следующей части этой главы я буду приводить примеры из nghhtp, поскольку этот инструмент наиболее прост и отлично подходит для использования в нашей книге. Независимо от того, какой инструмент вы используете, сообщения будут похожи. Однако порядок отображения данных или настройки могут слегка отличаться.

Данные инструменты полезны при отладке на низком уровне или просмотре деталей протокола, однако большинство людей не использует их постоянно каждый день. Разумеется, большинству разработчиков достаточно стандартных инструментов разработчика в браузерах, и им не придется углубляться в net-export нижнего уровня или уровень детализации Wireshark. Однако все три инструмента очень хороши, поскольку они легкодоступны и помогают закрепить ваше понимание протокола.

4.3.2 Формат фреймов HTTP/2

Прежде чем начать рассматривать примеры фреймов HTTP/2, необходимо понять, что они из себя представляют. Каждый фрейм состоит из

заголовка фиксированной длины (подробно описано в табл. 4.1) и информационного наполнения.

Таблица 4.1 Формат заголовков фреймов HTTP

Поле	Объем	Описание
Length	24 бит	Длина фрейма, не включая все поля заголовка, указанные в этой таблице, с максимальным размером 2^{24} – 1 октет; ограничена значением переменной SETTINGS_MAX_FRAME_SIZE, в которой по умолчанию задан наименьший размер 2^{14} октетов
Type	8 бит	В настоящее время определено 14 типов фреймов[a]: ▪ DATA (0x0); ▪ HEADERS (0x1); ▪ PRIORITY (0x2); ▪ RST_STREAM (0x3); ▪ SETTINGS (0x4); ▪ PUSH_PROMISE (0x5); ▪ PING (0x6); ▪ GOAWAY (0x7); ▪ WINDOW_UPDATE (0x8); ▪ CONTINUATION (0x9); ▪ ALTSVC(0xa), добавлен в RFC 7838[b]; ▪ (0xb), в настоящее время не используется[c]; ▪ ORIGIN (0xc), добавлен в RFC 8336[d]; ▪ CACHE_DIGEST, рекомендуемый[e]
Flags	8 бит	Флажки для конкретных фреймов
Reserved Bit	1 бит	В настоящее время не используется и должен быть установлен в значение 0
Stream Identifier	31 бит	Беззнаковое целое число, идентифицирующее фрейм

[a] https://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml.
[b] https://tools.ietf.org/html/rfc7838.
[c] https://github.com/httpwg/http-extensions/pull/323.
[d] https://tools.ietf.org/html/rfc8336.
[e] https://datatracker.ietf.org/doc/draft-ietf-httpbis-cache-digest/?include_text=1.

В HTTP/2 четко определены фреймы, что и делает его двоичным протоколом. Фиксированные фреймы HTTP/2 отличаются от текстовых HTTP/1-сообщений переменной длины, которые необходимо было анализировать путем сканирования разрывов строк и пробелов (малоэффективный процесс, в ходе которого существует вероятность возникновения ошибок). Строгий и четко определенный формат фреймов в HTTP/2 упрощает синтаксический анализ и сокращает объем сообщений благодаря использованию кодов (например, использование кода 0x01 для типа фрейма HEADERS вместо полной формулировки сообщения).

Октеты или байты?

В спецификации HTTP/2, как и во многих других официальных документах Internet-протоколов, используется понятие *октет*, а не байт, так как последнее допускает иные толкования. Октет составляет ровно 8 бит, но, в зависимости от архитектуры системы, байт может также составлять 8 бит.

Поле `Length`, как я надеюсь, не требует пояснений. В последующих разделах мы детально рассмотрим все типы фреймов. Поле `Flags` зависит от типа фрейма и описывается отдельно для каждого из них. Поле `Reserved Bit` в настоящее время не используется. Поле `Stream Identifier` также не требует особых объяснений. Вероятнее всего, причиной для ограничения его длины 31 битом стала совместимость с Java, поскольку данный язык не поддерживает 32-битное целое число без знака[1].

Значение флажков и информационное наполнение зависят от типа фрейма. HTTP/2 создан с возможностью дальнейшего расширения. В первоначальной версии спецификации было прописаны только фреймы 0–9[2], затем было добавлено еще три, и, несомненно, их количество будет расти.

4.3.3 Исследование потока сообщений HTTP/2 на примерах

При изучении фреймов HTTP/2 лучше всего рассматривать их использование на реальных примерах. К примеру, если мы перейдем на www.facebook.com (один из множества сайтов, поддерживающих HTTP/2) с помощью nghttp, мы получим следующий результат:

```
$ nghttp -va https://www.facebook.com | more
[  0.043] Connected
The negotiated protocol: h2
[  0.107] recv SETTINGS frame flags=0x00, stream_id=0>
          (niv=5) [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
[  0.107] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=10420225)
[  0.107] send SETTINGS frame <length=12, flags=0x00, stream_id=0>
          (niv=2)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
[  0.107] send SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=3>
          (dep_stream_id=0, weight=201, exclusive=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=5>
          (dep_stream_id=0, weight=101, exclusive=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=7>
          (dep_stream_id=0, weight=1, exclusive=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=9>
          (dep_stream_id=7, weight=1, exclusive=0)
```

[1] https://stackoverflow.com/questions/39309442/why-is-the-stream-identifier-31-bit-in-http-2-and-why-is-itpreceded-with-a-rese.

[2] https://tools.ietf.org/html/rfc7540.

```
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=11>
          (dep_stream_id=3, weight=1, exclusive=0)
[  0.107] send HEADERS frame <length=43, flags=0x25, stream_id=13>
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=11, weight=16, exclusive=0)
          ; Open new stream
          :method: GET
          :path: /
          :scheme: https
          :authority: www.facebook.com
          accept: */*
          accept-encoding: gzip, deflate
          user-agent: nghttp2/1.28.0
[  0.138] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
[  0.138] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=13>
          (window_size_increment=10420224)
[  0.257] recv (stream_id=13) :status: 200
[  0.257] recv (stream_id=13) x-xss-protection: 0
[  0.257] recv (stream_id=13) pragma: no-cache
[  0.257] recv (stream_id=13) cache-control: private, no-cache, no-store,
must-revalidate
[  0.257] recv (stream_id=13) x-frame-options: DENY
[  0.257] recv (stream_id=13) strict-transport-security: max-age=15552000;
Preload
[  0.257] recv (stream_id=13) x-content-type-options: nosniff
[  0.257] recv (stream_id=13) expires: Sat, 01 Jan 2000 00:00:00 GMT
[  0.257] recv (stream_id=13) set-cookie: fr=0m7urZrTka6WQuSGa..BaQ42y.61.A
AA.0.0.BaQ42y.AWXRqgzE; expires=Tue, 27-Mar-2018 12:10:26 GMT; Max-Age=7776 000;
path=/; domain=.facebook.com; secu
re; httponly
[  0.257] recv (stream_id=13) vary: Accept-Encoding
[  0.257] recv (stream_id=13) content-encoding: gzip
[  0.257] recv (stream_id=13) content-type: text/html; charset=UTF-8
[  0.257] recv (stream_id=13) x-fb-debug: yrE7eqv05dkxF8R1+i4VlIZmUNInVI+AP
DyG7HCW6t7NCEtGkIIRqJadLwj87Hmhk6z/N3O212zTPFXkT2GnSw==
[  0.257] recv (stream_id=13) date: Wed, 27 Dec 2017 12:10:26 GMT
[  0.257] recv HEADERS frame <length=517, flags=0x04, stream_id=13>
          ; END_HEADERS
          (padlen=0)
          ; First response header
<!DOCTYPE html>
<html lang="en" id="facebook" class="no_js">
<head><meta charset="utf-8" />
…и т. д.
[  0.243] recv DATA frame <length=1122, flags=0x00, stream_id=13>
…
[  0.243] recv DATA frame <length=2589, flags=0x00, stream_id=13>
…
```

```
[  0.264] recv DATA frame <length=13707, flags=0x00, stream_id=13>
…
[  0.267] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=33706)
[  0.267] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=13>
          (window_size_increment=33706)
…
[416.688] recv DATA frame <length=8920, flags=0x01, stream_id=13>
          ; END_STREAM
[417.226] send GOAWAY frame <length=8, flags=0x00, stream_id=0>
          (last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

В данном примере большинство фреймов `DATA` вырезано и заменено на `…и т. д.`, однако некоторые из них все же присутствуют.

Чтобы просмотреть только заголовки, поставьте в командной строке флажок `-n`:

```
$ nghttp -nv https://www.facebook.com | more
```

Несмотря на то что мы убрали информационную составляющую, код выглядит довольно сложным, поэтому мы расскажем вам о нем подробнее.

Сначала вы подключаетесь и согласовываете HTTP/2 через HTTPS (h2). Первым вы получаете фрейм `SETTINGS`, поскольку nghttp не выводит настройки HTTPS или преамбулу / «волшебное» сообщение HTTP/2:

```
$ nghttp -v https://www.facebook.com | more
[  0.043] Connected
The negotiated protocol: h2
[  0.107] recv SETTINGS frame <length=30, flags=0x00, stream_id=0>
          (niv=5)
          [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
```

Фрейм SETTINGS

Фрейм `SETTINGS` (`0x4`) – это первый фрейм, который отправляют сервер и клиент (после преамбулы / «волшебного» сообщения HTTP/2). Фрейм может быть или «пустышкой», или состоять из нескольких пар поле/значение, как показано в табл. 4.2.

Данный фрейм определяет только один флажок, который может быть установлен в общем заголовке: `ACK` (`0x1`). Если настройки соединения HTTP/2 устанавливаете вы, присвойте флажку значение `0`; для подтверждения настроек, установленных другой стороной, присвойте значение 1. При подтверждении (флажок имеет значение 1) нет необходимости устанавливать дополнительные настройки информационного наполнения.

Таблица 4.2 Формат фрейма HEADERS

Поле	Объем	Описание
Identifie	16 бит	В спецификации определены шесть настроек, и еще две были добавлены недавно (скорее всего, будут добавлены и другие). Предлагаемые настройки официально не стандартизированы: ▪ SETTINGS_HEADER_TABLE_SIZE (0x1); ▪ SETTINGS_ENABLE_PUSH (0x2); ▪ SETTINGS_MAX_CONCURRENT_STREAMS (0x3); ▪ SETTINGS_INITIAL_WINDOW_SIZE (0x4); ▪ SETTINGS_MAX_FRAME_SIZE (0x5); ▪ SETTINGS_MAX_HEADER_LIST_SIZE (0x6); ▪ SETTINGS_ACCEPT_CACHE_DIGEST (0x7)[a]; ▪ SETTINGS_ENABLE_CONNECT_PROTOCOL(0x8)[b]. Примечание: SETTINGS_ACCEPT_CACHE_DIGEST – это предполагаемый параметр, еще не стандартизованный официально и потенциально подверженный изменениям
Value	32 бита	В этом поле указывается значение параметра. Обратите внимание: если параметр не определен, используются значения по умолчанию. Предлагаемые настройки еще официально не стандартизированы: ▪ 4096 октетов; ▪ 1; ▪ Безлимитный; ▪ 65 535 октетов; ▪ 16 384 октетов; ▪ Безлимитный; ▪ 0 – Нет; ▪ 0 – Нет. Примечание: SETTINGS_ACCEPT_CACHE_DIGEST – это предполагаемый параметр, еще не стандартизованный официально и потенциально подверженный изменениям

[a] https://tools.ietf.org/html/draft-ietf-httpbis-cache-digest.
[b] https://tools.ietf.org/html/rfc8441.

Ознакомившись с информацией из таблицы, взгляните на первое сообщение еще раз:

```
[  0.107] recv SETTINGS frame <length=30, flags=0x00, stream_id=0>
          (niv=5)
          [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
```

Длина информационного наполнения полученного фрейма SETTINGS составляет 30 октетов, флажок подтверждения не установлен (поэтому фрейм подтверждения отсутствует), фрейм использует идентификатор потока 0. Идентификатор потока 0 зарезервирован для управляющих сообщений (фреймы SETTINGS и WINDOW_UPDATE), поэтому сервер и использует поток 0 для отправки данного кадра SETTINGS.

Затем вы получаете сами настройки. В данном примере их пять (niv=5), объем каждой из них составляет 16 (идентификатор) + 32 (значение) бит. Всего в примере 48 бит, или 6 октетов, что составляет длину, указанную в заголовке, – 30 октетов (5 заголовков × 6 октетов = 30 октетов). Все идет нормально. Теперь взгляните на индивидуальные настройки.

1 Значение параметра SETTINGS_HEADER_TABLE_SIZE у Facebook – 4069 октетов. Данный параметр используется для сжатия HTTP-заголовка HPACK, о котором мы поговорим в главе 8.

2 Также Facebook использует параметр SETTINGS_MAX_FRAME_SIZE со значением16 384 октета, благодаря чему вашему клиенту (nghttp) нет необходимости отправлять объемное информационное наполнение по этому соединению.

3 Затем Facebook устанавливает для параметра SETTINGS_MAX_HEADER_LIST_SIZE значение 131 072 октета, что запрещает клиенту отправлять объемные несжатые заголовки.

4 Сервер устанавливает параметр SETTINGS_MAX_CONCURRENT_STREAMS на 100 потоков. В главе 2 представлен пример, в котором была предпринята попытка загрузить более 100 изображений с сервера с ограничением в 100 потоков. В том случае запросы были поставлены в очередь в ожидании свободного потока, аналогично тому, как это происходит в очереди запросов в HTTP/1, однако с лимитом подключений ниже шести, что меньше чем у большинства браузеров. HTTP/2 позволяет значительно увеличить число выполняемых параллельных запросов, но зачастую их количество ограничивается сервером (обычно до 100 или 128 потоков), хотя по умолчанию ограничений нет.

5 Наконец, Facebook присваивает параметру SETTINGS_INITIAL_WINDOW_SIZE значение 65 536 октетов. Данный параметр используется для управления потоком, и мы поговорим о нем в главе 7.

В этом, казалось бы, простом фрейме присутствует несколько нюансов. Во-первых, настройки, описанные выше, могут идти в любом порядке, например SETTINGS_MAX_CONCURRENT_STREAMS, в спецификации определенный как настройка 3 (0x03), может идти после SETTINGS_MAX_HEADER_LIST_SIZE, который является настройкой 6 (0x06). Во-вторых, начальные значения многих параметров установлены по умолчанию, поэтому сервер может отправить сокращенный фрейм SETTINGS всего с тремя настройками, ничего при этом не теряя:

```
[  0.107] recv SETTINGS frame <length=18, flags=0x00, stream_id=0>
          (niv=3)
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
```

Однако в увеличении количества настроек нет ничего плохого, напротив, такой способ помогает указать более точные значения.

В данном примере мы видим, что значение параметра SETTINGS_INITIAL_WINDOW_SIZE на 1 октет больше, чем значение по умолчанию (65 535 октетов), что кажется слегка странным, поскольку нет необходимости менять значение по умолчанию.

К тому же обратите внимание, что сервер Facebook не устанавливает параметр SETTINGS_ENABLE_PUSH, который предназначен для отправки со-

общений сервером клиенту (используется на стороне клиента). Для сервера нет смысла устанавливать данный параметр, однако мы предполагаем, что его можно использовать для того, чтобы объявлять клиентам о том, что сервер поддерживает push-загрузку (если авторы спецификации решили использовать его для этой цели). В случаях, если клиент не поддерживает (или не хочет использовать) HTTP/2, лучше убрать этот параметр из фрейма SETTINGS.

Давайте рассмотрим еще один пример использования фрейма SETTINGS:

```
[  0.107] recv SETTINGS frame <length=30, flags=0x00, stream_id=0>
          (niv=5)
          [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
[  0.107] send SETTINGS frame <length=12, flags=0x00, stream_id=0>
          (niv=2)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
[  0.107] send SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
…
[  0.138] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
```

Сначала nghttp получает начальный фрейм SETTINGS сервера (уже знакомый нам), затем клиент отправляет фрейм SETTINGS с парой настроек. Далее клиент подтверждает фрейм SETTINGS сервера. Данный фрейм довольно прост, он имеет флажок ACK(0x01), длина составляет 0, и, следовательно, у него нет настроек (niv=0). Чуть ниже следует подтверждение сервером фрейма SETTINGS клиента в идентичном простом формате.

В данном примере существует отрезок времени, в течение которого одна из сторон уже отправила фрейм SETTINGS, но еще не получила ответ. В это время запрещено использовать настройки не по умолчанию. Однако такая ситуация не вызывает серьезных проблем, поскольку все реализации HTTP/2 должны иметь возможность обрабатывать значения по умолчанию, а также фрейм SETTINGS должен отправляться первым.

Фрейм WINDOW_UPDATE

Также сервер отправил фрейм WINDOW_UPDATE:

```
[  0.107] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=10420225)
```

Фрейм WINDOW_UPDATE (0x8) используется для управления потоком, например, с целью ограничения возможного количества отправленных

данных во избежание перегрузки получателя. В HTTP/1 одновременно может выполняться только один запрос. При перегрузке клиент он останавливает обработку TCP-пакетов; затем механизм управления TCP-потоком (аналогично управлению потоком HTTP/2) тормозит отправку данных до тех пор, пока получатель не будет готов обрабатывать новые данные. В HTTP/2 в одном и том же соединении может быть несколько потоков, и поэтому вы не зависите от управления TCP-потоком и должны реализовать свой собственный метод замедления для каждого потока.

Первоначальный размер отправляемого окна данных устанавливается в фрейме `SETTINGS`, а фрейм `WINDOW_UPDATE` применяется для увеличения его размера. Поэтому `WINDOW_UPDATE` является простым фреймом без каких-либо меток и с одним значением (и зарезервированным битом), как показано в табл. 4.3.

Таблица 4.3 Формат фрейма `WINDOW_UPDATE`

Поле	Объем	Описание
`Reserved Bite`	1 бит	Не используется
`Window Size Increment`	31 бит	Количество октетов, которое может быть отправлено до того, как должен быть получен следующий фрейм `WINDOW_UPDATE`

`WINDOW_UPDATE` не определяет флажки и применяется к текущему потоку или, если значение этого потока 0, применяется ко всему соединению HTTP/2. Поэтому отправители должны отслеживать как уровень потока, так и уровень соединения.

Управление потоком HTTP/2 применяется только к фреймам `DATA`. Все другие типы фреймов (или по крайней мере те, которые определены на данный момент) могут продолжать отправку, даже если места в окне управления потоком уже нет. Данная функция предотвращает блокировку важных управляющих сообщений (таких как само сообщение `WINDOW_UPDATE`) большими фреймами `DATA`. Кроме того, `DATA` является единственным фреймом, размер которого не ограничен.

Мы исследуем механизм управления потоком HTTP/2 в главе 7.

Фрейм PRIORITY

Далее следует несколько фреймов `PRIORITY` (`0x2`):

```
[  0.107] send PRIORITY frame frame <length=5, flags=0x00, stream_id=3>
          (dep_stream_id=0, weight=201, exclusive=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=5>
          ( dep_stream_id=0, weight=101, exclusive=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=7>
          (dep_stream_id=0, weight=1, exclusive=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=9>
          (dep_stream_id=7, weight=1, exclusive=0)
[  0.107] send PRIORITY frame <length=5, flags=0x00, stream_id=11>
          (dep_stream_id=3, weight=1, exclusive=0)
```

Данный код создает несколько потоков с различными приоритетами использования в nghttp. Фактически nghttp не использует потоки 3–11 напрямую; применяя параметр `dep-stream-id`, он «вешает» на них другие потоки, те, которые он устанавливал в самом начале. Использование предварительно созданных приоритетных потоков позволяет соответствующим образом приоритизировать запросы. Благодаря этому способу у вас отпадает необходимость устанавливать приоритеты для каждого последующего нового потока. Не все клиенты HTTP/2 используют этот подход, и nghttp основывает свою реализацию на модели Firefox[1], поэтому не беспокойтесь, если вы используете другой инструмент и не видите эти фреймы `PRIORITY`.

Приоритеты потоков HTTP/2 – довольно сложная тема, поэтому мы отложим ее рассмотрение до главы 7. На данном этапе имейте в виду, что некоторым запросам (например, исходному HTML, критическому CSS и критическому JavaScript) можно дать приоритет над менее важными запросами (такими как изображения или некритический асинхронный JavaScript). Формат таких фреймов представлен в табл. 4.4, но вы разберетесь в нем лучше после прочтения главы 7.

Таблица 4.4 Формат фреймов `PRIORITY`

Поле	Объем	Описание
`E (Exclusive)`	1 бит	Указывает, является ли поток эксклюзивным (устанавливается, только если для этого фрейма установлен флажок `Priority`)
`Stream Dependency`	31 бит	Индикатор того, от какого потока зависит этот заголовок (устанавливается, только если для этого кадра установлен флажок `Priority`)
`Weight`	8 бит	Вес этого потока (устанавливается, только если для этого фрейма установлен флажок `Priority`)

Фрейм `PRIORITY (0x2)` имеет фиксированную длину и не определяет флажки.

Фрейм HEADERS

Наконец, после установки всех настроек, мы переходим к прямому применению протокола и можем совершить запрос HTTP/2. Запросы отправляются в фрейме `HEADERS (0x1)`:

```
[  0.107] send HEADERS frame <length=43, flags=0x25, stream_id=13>
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=11, weight=16, exclusive=0)
          ; Open new stream
          :method: GET
          :path: /
          :scheme: https
          :authority: www.facebook.com
          accept: */*
          accept-encoding: gzip, deflate
          user-agent: nghttp2/1.28.0
```

[1] https://nghttp2.org/documentation/nghttp.1.html#dependency-based-priority.

Все строки, кроме первых двух, выглядят довольно похожими на запросы HTTP/1. Как вы, возможно, помните из главы 1, запрос HTTP/1 состоит из первой строки и обязательного заголовка хоста (и любых других заголовков HTTP):

```
GET / HTTP/1.1↵
Host: www.facebook.com↵
```

В HTTP/2 нет специального типа фреймов для запросов, а во фрейме HEADERS *каждая* строка отправляется в виде заголовка. Кроме того, для определения различных частей строки запроса HTTP создан ряд новых *псевдозаголовков* (строки, которые начинаются с двоеточия).

```
:method
: GET
:path: /
:scheme: https
:authority: www.facebook.com
```

Обратите внимание, что псевдозаголовок `:authority` заменил заголовок HTTP/1.1 `Host`. В отличие от стандартных HTTP-заголовков псевдозаголовки HTTP/2 строго определены[1], их нельзя добавить, не изменив протокол, и поэтому вы не сможете создать новый псевдозаголовок вроде этого:

```
:barry: value
```

Однако вы можете создавать обычные HTTP-заголовки. Для этого в любых заголовках, предназначенных специально для какого-то приложения, уберите начальное двоеточие:

```
barry: value
```

Новые спецификации все же позволяют создавать псевдозаголовки, и на момент написания данной книги такая спецификация существует. В документе *Bootstrapping WebSockets with HTTP/2 RFC*[2] был добавлен псевдозаголовок `:protocol`. Возможно, использование новых псевдозаголовков потребует обновления параметра SETTINGS, где необходимо будет указывать, поддерживает ли клиент или сервер такие новшества.

Такие псевдозаголовки могут отображаться в клиентских инструментах, например в инструментах разработчика Chrome (рис. 4.11), поэтому их наличие также указывает на то, что запрос совершен с помощью HTTP/2 (хотя на момент написания книги многие браузеры, такие как Firefox, их не отображают).

Стоит отметить, что HTTP/2 применяет строчные имена заголовков HTTP. Согласно официальной спецификации HTTP/1 данная версия не учитывала регистр для имен заголовков, хотя некоторые реализации и поступились этим правилом. Разные регистры могут быть в значениях заголовков, но не в их названиях. HTTP/2 не одобряет вольности в фор-

---

1 https://tools.ietf.org/html/rfc7540#section-8.1.2.

2 https://tools.ietf.org/html/rfc8441#section-3.

матировании заголовков. Излишние пробелы в начале строки, двойные двоеточия и символы возврата строки могут вызвать проблемы в работе протокола HTTP/2, несмотря на то что большинство реализаций HTTP/1 с этой проблемой справляется. Данный пример является одним из наиболее эффективных вариантов распознавания сообщений HTTP/2 на уровне фрейма. Описанные выше ошибки обычно возникают именно на этом уровне. Когда клиенты сталкиваются с недопустимыми заголовками, которые мешают работе вашего сайта, они выдают пользователю загадочные сообщения об ошибках (например, `ERR_SPDY_PROTOCOL_ERROR` в Chrome). Формат фрейма `HEADERS` представлен в табл. 4.5.

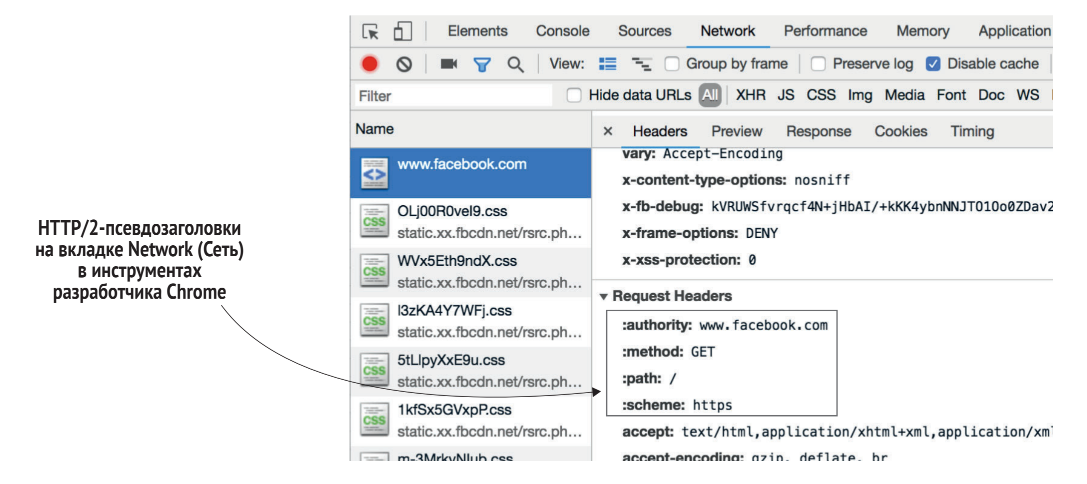


Рис. 4.11 Псевдозаголовки HTTP/2 в инструментах разработчика Chrome

Таблица 4.5 Формат фреймов `HEADERS`

Поле	Объем	Описание
`Pad Length`	8 бит (необязательно)	Необязательное поле, указывающее длину поля `Padding` (устанавливается, только если для этого фрейма установлен флажок `Padded`)
`E (Exclusive)`	1 бит	Указывает, является ли поток эксклюзивным (устанавливается, только если для этого фрейма установлен флажок `Priority`)
`Stream Dependency`	31 бит	Указывает поток, от которого зависит этот заголовок (устанавливается, только если для этого фрейма установлен флажок `Priority`)
`Weight`	8 бит	Вес этого потока (устанавливается, только если для этого фрейма установлен флажок `Priority`)
`Header Block Fragment`	Объем фрейма за вычетом других полей в этой таблице	Заголовки запроса (включая псевдозаголовки)
`Padding`	Определяется полем `Pad Length` (необязательно)	Устанавливает значение `0` для каждого заполняемого байта (устанавливается, только если для этого фрейма установлен флажок `Padded`)

Поля `E`, `Stream Dependency` и `Weight` мы обсудим в главе 7. Поля `Pad Length` и `Padding` добавлены по соображениям безопасности. При необходимости

они позволяют скрыть истинную длину сообщения. В поле `Header Block Fragment` отправляются все заголовки (включая псевдозаголовки). Как можно судить по выводу nghhttp, это поле не является открытым текстом. В главе 8 мы рассмотрим формат сжатия заголовков HPACK, поэтому сейчас вам не следует беспокоиться об этом, к тому же, такие инструменты, как nghhttp, автоматически распаковывают заголовки HTTP за вас.

Фрейм `HEADERS` использует четыре флажка, которые могут быть установлены в общем заголовке фрейма:

- `END_STREAM (0x1)` устанавливается при условии, что за фреймом `HEADERS` не следуют никакие другие фреймы (например, фрейм `DATA` для запроса `POST`). Парадоксально, но фреймы `CONTINUATION` (которые мы обсудим далее в этой главе) не ограничиваются как `HEADERS`; они считаются продолжением фрейма `HEADERS`, а не дополнительными фреймами, и управляются флажком `END_HEADERS`;
- `END_HEADERS (0x4)` указывает, что все заголовки HTTP включены во фрейм и за ними не следует фрейм `CONTINUATION`, в котором могут содержаться дополнительные заголовки;
- `PADDED (0x8)` устанавливается, когда используется заполнение. Она означает, что первые 8 бит фрейма `DATA` указывают, насколько был заполнен фрейм `HEADERS`;
- `PRIORITY (0x20)` указывает, что во фрейме установлены поля `E`, `Stream Dependency` и `Weight`.

Если HTTP-заголовок состоит из более чем одного фрейма, вторым фреймом станет не дополнительный фрейм `HEADERS`, а фрейм `CONINTUATION`, следующий сразу за `HEADERS`. В сравнении с телами HTTP, которые используют столько фреймов `DATA`, сколько потребуется, данный процесс может показаться чрезмерно усложненным. Но другие поля, представленные в табл. 4.5, можно использовать только один раз. Если устанавливать их в последующих фреймах `HEADERS` для одного и того же запроса, могут возникнуть некоторые проблемы. Поэтому мультиплексированный характер HTTP/2 ограничивается требованием, согласно которому фреймы `CONTINUATION` должны следовать сразу за `HEADERS`, а их чередование запрещено. Именно поэтому были рассмотрены некоторые альтернативы[1]. Сейчас фрейм `CONTINUATION` используется довольно редко, и большинство запросов умещается в один фрейм `HEADERS`.

Теперь, владея вышеизложенной информацией, вы сможете лучше понять первую часть сообщения:

```
[  0.107] send HEADERS frame <length=43, flags=0x25, stream_id=13>
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=11, weight=16, exclusive=0)
          ; Open new stream
          :method: GET
          :path: /
          :scheme: https
          :authority: www.facebook.com
```

[1] https://github.com/http2/http2-spec/wiki/ContinuationProposals.

```
accept: */*
accept-encoding: gzip, deflate
user-agent: nghttp2/1.28.0
```

Каждому новому запросу присваивается уникальный идентификатор потока, значение которого больше, чем у идентификатора последнего используемого потока (в данном случае идентификатор последнего фрейма `PRIORITY`, созданного ngix, имеет значение 11, а новому фрейму присваивается значение 13, поскольку четное значение зарезервировано за сервером). Также устанавливаются различные метки, которые в совокупности образуют шестнадцатеричное значение `0x25`, а nghttp2 помогает отображать их в строке ниже. Метки `END_STREAM` (`0x1`) и `END_HEADERS` (`0x4`) указывают на то, что фрейм содержит полный запрос и не имеет фрейма `DATA` (что встречается у запросов `POST`). Метка `PRIORITY` (`0x20`) указывает на приоритизацию. Сложив все эти шестнадцатеричные значения (`0x1` + `0x4` + `0x20`), вы получите `0x25`, как и указано в заголовке фрейма. Данный поток зависит от идентификатора потока 11, поэтому ему присваивается соответствующий приоритет и вес 16 в рамках этого приоритета. Я вновь прошу вас не особо задумываться над этим сейчас, поскольку о приоритизации мы подробнее поговорим в главе 7. Инструмент nghttp отмечает, что этот поток новый (`Open new stream`), а затем следует перечисление различных псевдозаголовков HTTP и заголовков запросов HTTP.

HTTP-ответы содержат фрейм `HEADERS` в том же потоке, как вы можете увидеть в следующем примере:

```
[  0.257] recv (stream_id=13) :status: 200
[  0.257] recv (stream_id=13) x-xss-protection: 0
[  0.257] recv (stream_id=13) pragma: no-cache
[  0.257] recv (stream_id=13) cache-control: private, no-cache, no-store,
must-revalidate
[  0.257] recv (stream_id=13) x-frame-options: DENY
[  0.257] recv (stream_id=13) strict-transport-security: max-age=15552000;
preload
[  0.257] recv (stream_id=13) x-content-type-options: nosniff
[  0.257] recv (stream_id=13) expires: Sat, 01 Jan 2000 00:00:00 GMT
[  0.257] recv (stream_id=13) set-cookie: fr=0m7urZrTka6WQuSGa..BaQ4Ay.61.A
AA.0.0.BaQ42y.12345678; expires=Tue, 27-Mar-2018 12:10:26 GMT; Max-Age=7776
000; path=/; domain=.facebook.com; secu
re; httponly
[  0.257] recv (stream_id=13) set-cookie: sb=so11234567890TZ7e-i5S2To; expi
res=Fri, 27-Dec-2019 12:10:26 GMT; Max-Age=63072000; path=/; domain=.facebo
ok.com; secure; httponly
[  0.257] recv (stream_id=13) vary: Accept-Encoding
[  0.257] recv (stream_id=13) content-encoding: gzip
[  0.257] recv (stream_id=13) content-type: text/html; charset=UTF-8
[  0.257] recv (stream_id=13) x-fb-debug: yrE7eqv05dkxF8R1+1234567890nVI+AP
DyG7HCW6t7NCEtGkIIRqJadLwj87Hmhk6z/N3O212zTPFXkT2GnSw==
[  0.257] recv (stream_id=13) date: Wed, 27 Dec 2017 12:10:26 GMT
[  0.257] recv HEADERS frame <length=517, flags=0x04, stream_id=13>
          ; END_HEADERS
          (padlen=0)
          ; First response header
```

В данном примере сначала идет псевдозаголовок (`:status: 200`), который, в отличие от того же процесса в HTTP/1.1, вместо текстового представления кода состояния (например, `200 OK`) дает только трехзначное числовое значение (`200`). За ним следуют различные HTTP-заголовки, которые отправляются немного иначе (мы рассмотрим это при обзоре HPACK в главе 8). Затем nghttp перечисляет параметры фрейма `HEADERS`. Как ни странно, nghttp предоставляет параметры фрейма после информационного наполнения фрейма, а не до него, что было бы предпочтительнее[1]. Параметры включают в себя флажок `END_HEADERS (0x04)`, который сигнализирует, что весь заголовок HTTP-ответа помещается в один фрейм `HEADERS`.

Завершающие заголовки

В HTTP/1.1 была впервые представлена концепция завершающих заголовков, которые можно отправлять после тела запроса. Они позволяют использовать метаинформацию, которая не может быть рассчитана заранее. Например, для потоковых данных можно вычислить контрольную сумму или цифровую подпись контента и включить ее в завершающий HTTP-заголовок.

В реальности завершающие заголовки не поддерживаются в большинстве реализаций и используются крайне редко. Но в HTTP/2 они поддерживаются, поэтому фрейм `HEADERS` (или фрейм `HEADERS`, за которым следует один или несколько фреймов `CONTINUATION`) появляется до и (необязательно) после всех фреймов `DATA` для этого потока.

Фрейм DATA

За фреймом `HEADERS` следует фрейм `DATA (0x0)`, который используется для отправки тела сообщения. В HTTP/1.0 тело сообщения следовало после заголовков и двух символов разрыва строки (сигнализирующих об окончании заголовка). В HTTP/2 данные оформлены в отдельный тип сообщения. Вы можете отправлять заголовки, за которыми следует часть тела, затем часть другого потока, снова часть тела и т. д. Разделив ответы HTTP/2 на один или несколько фреймов, вы можете мультиплексировать потоки через одно и то же соединение.

Фреймы `DATA` в HTTP/2 довольно просты и содержат различные данные, которые могут понадобиться на странице: текст в кодировке UTF-8, сжатые с помощью gzip файлы, код HTML, байты, составляющие изображение JPEG – что угодно. Заголовок основного фрейма содержит длину, поэтому данное значение не требуется для фрейма `DATA`. Как и фрейм `HEADERS`, фрейм `DATA` позволяет использовать заполняющие данные (паддинг), чтобы скрыть размер сообщения по соображениям безопасности, поэтому в начале может присутствовать поле `Pad Length` для указания длины. Насколько проста структура фрейма `DATA`, вы можете увидеть в табл. 4.6.

---

[1] https://github.com/nghttp2/nghttp2/issues/1163.

Таблица 4.6 Формат фрейма DATA

Поле	Объем	Описание
Pad Length	8 бит (необязательно)	Необязательное поле, указывающее длину поля Padding (присутствует, только если установлен флажок PADDED)
Data	Объем фрейма за вычетом полей Padding	Данные
Padding	Определяется полем Pad Length (необязательно)	Установка значения 0 для каждого заполняемого байта (присутствует, только если установлен флажок PADDED)

Фрейм DATA определяет два флажка, которые могут быть установлены в заголовке фрейма:

- END_STREAM (0x1) устанавливается, если кадр является последним в потоке;
- PADDED (0x8) устанавливается при использовании заполнения. Это означает, что первые 8 бит кадра DATA используются для обозначения степени заполнения в конце кадра.

В следующем примере большая часть содержимого вырезана из соображений экономии места; обычно строки ест... заполняются соответствующими данными:

```
<!DOCTYPE html>
<html lang="en" id="facebook" class="no_js">
<head><meta charset="utf-8" />
…и т. д.
[  0.243] recv DATA frame <length=1122, flags=0x00, stream_id=13>
…и т. д.
[  0.243] recv DATA frame frame <length=2589, flags=0x00, stream_id=13>
…и т. д.
[  0.264] recv DATA frame <length=13707, flags=0x00, stream_id=13>
…и т. д.
[  0.267] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=33706)
[  0.267] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=13>
          (window_size_increment=33706)
…и т. д.
[416.688] recv DATA frame <length=8920, flags=0x00, stream_id=13>
```

Здесь мы видим, как HTML-код отправляется в различных фреймах DATA (nghttp помогает вам распаковать данные), и, когда клиент обрабатывает эти фреймы, он в ответах отправляет фреймы WINDOW_UPDATE, позволяя серверу продолжать отправлять больше данных. Интересно, что Facebook по умолчанию отправляет относительно небольшие кадры DATA (1122 октета, 2589 октетов и т. д.), несмотря на то что клиент может обрабатывать кадры гораздо большего размера (до 65 535 октетов). Я не уверен, что такой способ выбран преднамеренно (чтобы как можно быстрее передать как можно больше данных клиенту). Возможно, так происходит, поскольку окно перегрузки TCP изначально невелико или по какой-либо другой причине.

Поскольку фреймы DATA в HTTP/2 по умолчанию могут быть разделены на части, нет необходимости в специальном механизме фрагментиро-

ванной передачи (обсуждается в разделе 4.1.1). В спецификации HTTP/2 сказано: «Механизм фрагментированной передачи … НЕ ДОЛЖЕН использоваться в HTTP/2».

Фрейм GOAWAY

Фрейм GOAWAY (0x7) выглядит следующим образом:

```
[417.226] send GOAWAY frame <length=8, flags=0x00, stream_id=0>
          (last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

Этот фрейм с несколько грубым названием (*go away* – «убирайся прочь») используется для закрытия соединений по двум причинам: либо потому, что сообщений, которые могут быть отправлены, больше нет, либо потому, что возникла серьезная ошибка. В табл. 4.7 приведен формат фрейма GOAWAY.

Таблица 4.7 Формат фрейма GOAWAY

Поле	Объем	Описание
Reserved Bit	1 бит	Не используется
Last-Stream-ID	31 бит	ID последнего обработанного входящего потока, чтобы клиент мог знать, был ли пропущен недавно инициированный поток
Error Code	32 бита	Код ошибки, если фрейм GOAWAY был отправлен из-за ошибки: ▪ NO_ERROR(0x0); ▪ PROTOCOL_ERROR(0x1); ▪ INTERNAL_ERROR(0x2); ▪ FLOW_CONTROL_ERROR(0x3); ▪ SETTINGS_TIMEOUT(0x4); ▪ STREAM_CLOSED(0x5); ▪ FRAME_SIZE_ERROR(0x6); ▪ REFUSED_STREAM(0x7); ▪ CANCEL(0x8); ▪ COMPRESSION_ERROR(0x9); ▪ CONNECT_ERROR(0xa); ▪ ENHANCE_YOUR_CALM(0xb); ▪ INADEQUATE_SECURITY(0xc); ▪ HTTP_1_1_REQUIRED(0xd)
Additional Debug Data	Остаток длины кадра (необязательно)	Неопределенный, зависящий от реализации формат

Фрейм GOAWAY не определяет флажки.

Посмотрев на последнее сообщение в предыдущем выводе nghttp, вы можете увидеть пример фрейма GOAWAY:

```
[417.226] send GOAWAY frame <length=8, flags=0x00, stream_id=0>
          (last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

Клиент nghttp сам отправляет фрейм GOAWAY, а не получает его от сервера. В этом примере nghttp получает HTML-код домашней страницы и не запрашивает все зависимые ресурсы (CSS, JavaScript и т. д.), которые запрашивал бы обычный браузер. После того как клиент получает и обрабатывает все данные, он отправляет этот фрейм для закрытия соединения HTTP/2. Веб-браузеры в большинстве случаев оставляют соединение открытым на случай возникновения последующих запросов, однако

использование nghttp завершается после получения одного такого ответа, поэтому перед выходом закрывается и соединение. При закрытии сеанса браузера то же самое может происходить с любыми открытыми соединениями.

Фрейм был отправлен с минимальной длиной в 8 октетов (1 бит + 31 бит + 32 бита); флажки не устанавливались; фрейм отправлен в потоке `0`. Последний идентификатор потока, полученный от сервера, имел значение `0`, поэтому потоков, инициированных сервером, не было. Также не было кодов ошибок (`NO_ERROR [0x00]`) и дополнительных отладочных данных. Таким образом, этот пример представляет собой стандартный способ аккуратно закрыть соединение, когда в нем уже нет необходимости.

4.3.4 Дополнительные фреймы

На примере Facebook мы рассмотрели многие типы фреймов HTTP/2, однако существует еще несколько типов, не вошедших в простой поток из примера. Кроме того, протокол HTTP/2 предусматривает возможность добавления в него новых фреймов. Недавно были добавлены три новых типа фреймов – `ALTSVC`, `ORIGIN` и `CACHE_DIGEST`, – которые мы обсудим в конце данного раздела. На момент написания этой книги формально стандартизированы только два типа, а последний, возможно, будет стандартизирован к моменту публикации. Каждый новый фрейм, настройка HTTP/2 или код ошибки должны быть зарегистрированы в Internet Assigned Numbers Authority (IANA)[1].

Фрейм CONTINUATION

Фрейм `CONTINUATION (0x9)` используется для больших HTTP-заголовков и следует сразу за фреймом `HEADERS` или `PUSH_PROMISE`. Поскольку весь заголовок должен быть доступен до того, как запрос может быть обработан, а также для контроля словаря HPACK (см. главу 8), фрейм `CONTINUATION` должен следовать сразу за фреймом `HEADERS`, нуждающимся в продолжении. Как уже упоминалось ранее при обсуждении фрейма `HEADERS`, данное требование ограничивает возможности мультиплексирования потока HTTP/2. Возникало много споров о необходимости использования фрейма `CONTINUATION` и о том, следует ли разрешить больший размер фреймов `HEADERS`. Пока `CONTINUATION` еще используется, однако вряд ли у него есть хорошие перспективы. Фрейм `CONTINUATION` проще, чем фреймы `HEADER` или `PUSH_PROMISE`, продолжением которых он является. Он содержит дополнительные данные заголовка. Формат данного фрейма приведен в табл. 4.8.

Таблица 4.8 Формат фрейма CONTINUATION

Поле	Длина	Описание
Header Block Fragment	Длина кадра без этого поля	Информация

[1] https://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml.

Фрейм CONTINUATION определяет только один флажок, который может быть установлен в общем заголовке фрейма. Флажок END_HEADERS (0x4), если он установлен, указывает, что в данном фрейме завершены все заголовки и за ним не следует фрейм CONTINUATION с дополнительными заголовками.

Фрейм CONTINUATION не использует флажок END_STREAM для указания на отсутствие тела, поскольку является продолжением исходного фрейма HEADERS.

Фрейм PING

Фрейм PING (0x6) используется для приблизительного измерения объема данных, проступающих со стороны отправителя, а также может использоваться для поддержания активности неиспользуемого соединения. Получив этот фрейм, принимающая сторона должна немедленно ответить аналогичным фреймом PING. Оба фрейма должны быть отправлены в потоке управления (идентификатор потока 0). Формат фрейма PING приведен в табл. 4.9.

Таблица 4.9 Формат фрейма PING

Поле	Длина	Описание
Opaque Data	64 бита (8 октетов)	Данные, которые будут отправлены в ответном запросе PING

Фрейм PING определяет одну метку, которая может быть установлена в общем заголовке фрейма. ACK (0x1) не следует устанавливать в отправляемом фрейме PING, но он должен быть в возвращаемом фрейме PING.

Фрейм PUSH_PROMISE

Фрейм PUSH_PROMISE (0x5) используется сервером, чтобы сообщить клиенту, что сервер собирается отправить ресурс, который клиент не запрашивал. Фрейм PUSH_PROMISE должен предоставить клиенту информацию о ресурсе, который будет отправлен, поэтому он включает все заголовки HTTP, которые обычно включаются в запрос фрейма HEADERS (и аналогично может сопровождаться фреймом CONTINUATION для push-запросов с заголовками, превышающими размер одного фрейма). Формат фрейма PUSH_PROMISE приведен в табл. 4.10.

Таблица 4.10 Формат фрейма PUSH_PROMISE

Поле	Длина	Описание
Pad Length	8 бит (необязательно)	Необязательное поле, указывающее длину поля Padding
Reserved Bit	1 бит	Не используется
Promised Stream ID	31 бит	Указывает поток, в котором будет отправлено push-обещание
Header Block Fragment	Длина фрейма за вычетом других полей в этой таблице	Заголовки HTTP отправленного ресурса
Padding	Определяется полем Pad Length (необязательно)	Устанавливает в 0 все заполняющие байты

Фрейм PUSH_PROMISE определяет два флажка, которые могут быть установлены в общем заголовке фрейма:

- END_HEADERS (0x4) указывает, что все заголовки содержатся в этом фрейме и за ними не следует фрейм CONTINUATION с дополнительными заголовками;
- PADDED (0x8) устанавливается при использовании заполнения. Это означает, что первые 8 бит фрейма DATA используются для обозначения объема заполнения, которое было добавлено в конец фрейма PUSH_PROMISE.

Мы обсудим механизм server push протокола HTTP/2 в главе 5.

Фрейм RST_STREAM

Последний фрейм, определенный в исходной спецификации HTTP/2, – это фрейм RST_STREAM (0x3), который используется для немедленной отмены (или сброса) потока. Необходимость в отмене может возникнуть ввиду ошибки или ввиду того, что запрос уже не требуется. Также она возникает, когда клиент покидает соединение, отменяет загрузку или ему больше не нужен выгружаемый сервером ресурс.

HTTP/1.1 не поддерживает эту функцию. Если вы начнете загружать объемный ресурс на какой-либо странице, вы не сможете отключить его загрузку, даже покидая страницу. Единственным выходом является закрытие соединения. У вас нет возможности отменить запрос, если он находится в активном состоянии. Данная функция – это еще одно преимущество HTTP/2 перед HTTP/1.1. Формат фрейма RST_STREAM приведен в табл. 4.11.

Таблица 4.11 Формат фрейма RST_STREAM

Поле	Длина	Описание
Error Code	32 бита	Код ошибки, указывающий на причину прерывания потока: ▪ NO_ERROR (0x0); ▪ PROTOCOL_ERROR (0x1); ▪ INTERNAL_ERROR (0x2); ▪ FLOW_CONTROL_ERROR (0x3); ▪ SETTINGS_TIMEOUT (0x4); ▪ STREAM_CLOSED (0x5); ▪ FRAME_SIZE_ERROR (0x6); ▪ REFUSED_STREAM (0x7); ▪ CANCEL (0x8); ▪ COMPRESSION_ERROR (0x9); ▪ CONNECT_ERROR (0xa); ▪ ENHANCE_YOUR_CALM (0xb); ▪ INADEQUATE_SECURITY (0xc); ▪ HTTP_1_1_REQUIRED (0xd)

Фрейм RST_STREAM не определяет флажки.

В спецификации содержится мало информации о данных кодах ошибок, и даже если она есть, то не всегда вполне ясна. Например, в спецификации сказано следующее насчет того, что один из двух кодов ошибок может использоваться для отмены отправленных ответов:

> *Если по какой-либо причине клиент отказывается от получения уже отправленного ответа от сервера или если серверу требуется слишком много времени на отправку обещанного ответа, клиент может отправить фрейм* RST_STREAM *и указать код* CANCEL *или* REFUSED_STREAM*, а также ссылку на идентификатор отправляемого потока.*

В конечном итоге разработчики сами решают, какие коды ошибок использовать и когда. Реализации не всегда поддерживают использование выбранных вами кодов.

Фрейм ALTSVC

Фрейм ALTSVC (0xa) был первым фреймом, добавленным к HTTP/2 после утверждения спецификации. Он подробно описан в отдельной спецификации[1] и позволяет серверу объявлять об использовании альтернативных служб, доступных для получения этого ресурса, как описано в разделе 4.2.4. Данный фрейм можно использовать для обновления протокола (например, до соединения h2c с h2) или для направления трафика на другую версию (см. табл. 4.12).

Таблица 4.12 Формат фрейма ALTSVC

Поле	Длина	Описание
Origin-Len	16 бит	Длина поля Origin
Origin	Определяется полем Origin-Len (необязательно)	Альтернативный URL
Alt-Svc-Field-Value	Длина фрейма за вычетом других полей в этой таблице	Альтернативный вид сервиса

Фрейм ALTSVC не определяет флажки.

Фрейм ORIGIN

Фрейм ORIGIN (0xc) – это новый фрейм, введенный в спецификацию в марте 2018 года2. Он позволяет серверу указывать, исходя из какой начальной точки (например, имя домена) этот сервер будет отвечать. Данный фрейм полезен для клиента, поскольку помогает решить, объединять ли уже установленные соединения с этим соединением HTTP/2. Формат фрейма ORIGIN представлен в табл. 4.13.

Таблица 4.13 Формат фрейма ORIGIN

Поле	Длина	Описание
Origin-Len	16 бит	Длина поля Origin
Origin	Обозначается полем Origin-Len (необязательно)	Альтернативный URL

Во всю длину фрейма можно включить несколько пар Origin-Len/Origin. Фрейм ORIGIN не определяет флажки. При обсуждении объединения соединений в главе 6 мы еще вернемся к фрейму ORIGIN.

---

[1] https://tools.ietf.org/html/rfc7838.

[2] https://tools.ietf.org/html/rfc8336.

Фрейм CACHE_DIGEST

Фрейм CACHE_DIGEST(0xd) на момент написания книги имел статус нового предложения, находящегося на утверждении[1]. Он позволяет клиенту указать, какие ресурсы он кешировал. Это означает, что серверу не следует отправлять эти ресурсы, так как они уже есть у клиента. Формат фрейма CACHE_DIGEST на момент написания книги (он может измениться) приведен в табл. 4.14.

Таблица 4.14 Формат фрейма CACHE_DIGEST

Поле	Длина	Описание
Origin-Len	16 бит	Длина поля Origin
Origin	Определяется полем Origin-Len (необязательно)	То, к чему относится адрес начала краткого изложения
Digest-Value	Длина кадра за вычетом других полей в этой таблице (необязательно)	Cache-Digest (обсуждается в главе 5)

Фрейм CACHE_DIGEST определяет следующие флажки:

- RESET (0x1) дает указание серверу сбросить любую текущую информацию CACHE_DIGEST;
- COMPLETE (0x2) указывает, что включенные дайджесты являются не сегментом кеша, а его полным представлением.

При обсуждении механизма push-загрузки протокола HTTP/2 в главе 5 мы еще вернемся к фрейму CACHE_DIGEST.

Резюме

- HTTP/2 – это двоичный протокол, в котором сообщения имеют особый формат и структуру.
- По этой причине перед отправкой любых HTTP-сообщений клиент и сервер должны подтвердить использование HTTP/2.
- Веб-браузеры согласовывают этот аспект посредством установки HTTPS-соединения с использованием нового расширения под названием ALPN.
- В HTTP/2 запросы и ответы отправляются и принимаются в формате фреймов.
- Например, запрос HTTP/2 GET обычно отправляется в фрейме HEADERS, а ответ обычно выглядит как фрейм HEADERS, за которым следуют фреймы DATA.
- У большинства веб-разработчиков и администраторов веб-серверов нет необходимости просматривать фреймы HTTP/2, хотя существуют инструменты, с помощью которых это можно сделать.
- Уже утвержден довольно обширный набор фреймов HTTP/2 и, кроме того, можно добавлять новые.

[1] https://tools.ietf.org/html/draft-ietf-httpbis-cache-digest.

5

Реализация HTTP/2 push

В этой главе мы рассмотрим:

- что такое HTTP/2 push;
- различные способы запроса HTTP/2 push;
- работа HTTP/2 push со стороны сервера и клиента;
- где HTTP/2 push применяется, а где нет;
- устранение неполадок HTTP/2 push;
- некоторые риски HTTP/2 push.

5.1 Что такое HTTP/2 server push?

Концепция push-загрузки в HTTP/2 (далее HTTP/2 push) позволяет серверам вместе с ответами отправлять дополнительные ресурсы, которые не были запрошены клиентом. До появления версии HTTP/2 HTTP был простым протоколом запроса и ответа; браузер запрашивал ресурс, и сервер отвечал ему этим ресурсом. Если требовались дополнительные ресурсы (такие как CSS, JavaScript, шрифты, изображения и т. д.), браузеру приходилось загружать начальную страницу, проверять наличие ссылок на дополнительные ресурсы, а затем запрашивать их. В случае изображений выполнение дополнительных запросов не представляло серьезной проблемы, так как их загрузка обычно не задерживает визуализацию страницы, поскольку вместо них во время старта рендеринга может быть просто пустое место. Однако некоторые ресурсы критически важны для рендеринга страницы (например, CSS и JavaScript), и браузер даже не будет пытаться отобразить страницу, пока эти ресурсы не загру-

жены. Такой процесс добавляет по крайней мере один дополнительный цикл приема и передачи, и поэтому замедляет просмотр веб-страниц. Мультиплексирование HTTP/2 позволяет запрашивать все ресурсы параллельно в одном соединении, что, в отличие от HTTP/1, помогает сократить очереди. Без HTTP/2 push браузеру пришлось бы совершать дополнительные запросы только после загрузки начальной страницы. Следовательно, в лучшем случае для большинства запросов веб-страниц требовалось бы не менее двух циклов приема-передачи. На рис. 5.1 показано, что файл CSS и файл JavaScript загружаются одновременно во втором наборе запросов.

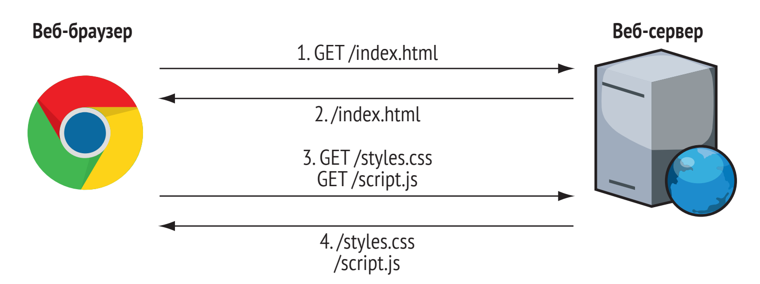


Рис. 5.1 Дополнительный цикл приема-передачи при загрузке критических ресурсов

На рис. 5.2 мы видим, что для начала визуализации страницы требуется два цикла. Обратите внимание, загрузка ресурсов styles.css и script.js происходит в разное время из-за ограничений сети или обработки; они загружаются не параллельно.

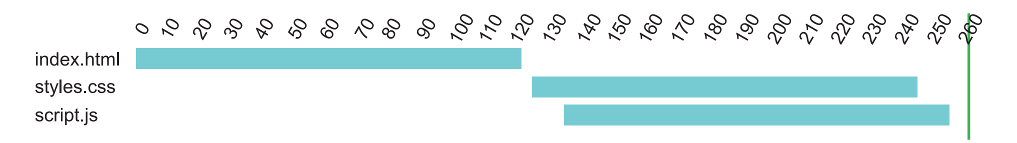


Рис. 5.2 Задержка приема-передачи критических ресурсов в виде каскадной диаграммы

По причине этой задержки появилась необходимость в оптимизации производительности; например, теперь таблицы стилей встраиваются непосредственно в HTML-страницу с помощью тегов `<style>`, а в JavaScript подобные процессы совершаются с помощью тегов `<script>`. Таким образом, браузеры могут запускать процесс рендеринга сразу после загрузки и анализа начальной страницы и не ждать загрузки дополнительных критических ресурсов.

Однако встраивание ресурсов также имеет некоторые недостатки. Для CSS встраиваются только критически важные части стиля (те, что необходимы для запуска рендеринга страницы), а полный файл стилей загружается позже, что позволяет минимизировать количество встроенного кода и сократить размер страницы. Извлечь необходимые стили из

ресурсов CSS и встроить их в HTML-файл довольно сложно, однако и для этой задачи существуют свои инструменты. Данный процесс не только сложный, но еще и довольно ресурсоемкий; критический CSS хранится не в одном CSS-файле, который можно кешировать и использовать на других страницах, а дублируется на каждой странице веб-сайта. Кроме того, встраиваемый критический CSS находится и в основном CSS-файле, который загружается позже. Таким образом, он дублируется на каждой странице! Также процесс встраивания требует использовать JavaScript для загрузки некритических файлов CSS. Для этого используется стандартный тег `<link rel="stylesheet" type="text/css" href="…">`, что приводит к приостановке рендеринга до загрузки файлов, так как для тегов CSS `link` нет атрибута `async`. Кроме того, если вы захотите внести изменения в критический CSS (например, при редизайне сайта), вместо одного общего CSS-файла вам нужно будет вносить изменения на каждую страницу. В общем и целом встраивание обеспечивает неплохой прирост производительности при первом посещении сайта, однако это все же своего рода ухищрение. HTTP/2 push является другим (и лучшим) способом решения данной проблемы.

HTTP/2 push исключает парадигму «один запрос – один ответ», согласно которой всегда работал протокол HTTP. Push-загрузка позволяет серверу на один запрос дать несколько ответов. На вопрос «Скажите, можно ли мне получить эту страницу?» вполне может последовать ответ: «Да, конечно, и вот вам еще несколько ресурсов, которые могут пригодиться для этой страницы». Рисунок 5.3 показывает, что для получения страницы и критических ресурсов, необходимых для старта рендеринга, можно обойтись одним циклом «запрос–ответ».

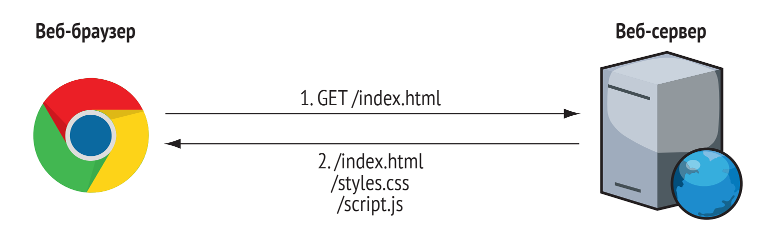


Рис. 5.3 Использование HTTP/2 push может устранить задержку приема-передачи для критических ресурсов

Данный процесс можно представить и в виде каскадной диаграммы, как на рис. 5.4. На нем видно, что получение ресурсов происходит не одновременно, о чем свидетельствуют короткие промежутки между ними. Однако на это требуется времени лишь немного больше одного цикла, а не два полноценных цикла «запрос–ответ».

Также экономию времени можно увидеть на диаграммах типа «запрос–ответ», представленных в главе 2. На рис. 5.5 видно, что время значительно экономится за счет того, что все критические ресурсы отправляются вместе с начальной страницей.

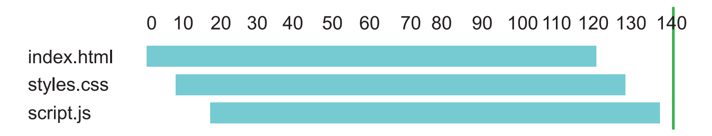


Рис. 5.4 Каскадная диаграмма использования HTTP/2 push для получения всех запросов за один цикл приема-передачи

Время (мс)
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250 260 270

Запрос веб-страницы
Получение запроса веб-страницы
Отправка HTML-страницы
Получение HTML-страницы
Запрос CSS
Запрос JS
Получение запроса CSS
Получение запроса JS
Отправка CSS
Отправка JS
Запрос CSS
Запрос JS
Прорисовка страницы

Время (мс)
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250 260 270

Запрос веб-страницы
Получение запроса веб-страницы
Отправка HTML-страницы
Отправка CSS
Отправка JS
Получение HTML-страницы
Запрос CSS
Запрос JS
Прорисовка страницы

Рис. 5.5 Поток запросов для базовой веб-страницы без HTTP/2 push (слева) и с HTTP/2 push (справа)

HTTP/2 push помогает сократить время загрузки, но только при условии правильного использования. Загрузка слишком большого количества ресурсов *(перегрузка)*, которые клиент не будет использовать или которые уже есть в его кеше, может привести к задержкам. Вместо того чтобы загрузить *действительно нужный вам ресурс*, пропускная способность соединения будет использоваться для загрузки не столь важных ресурсов. HTTP/2 push следует использовать вдумчиво и с осторожностью, о чем мы и поговорим в этой главе.

Может ли HTTP/2 push заменить WebSockets или SSE?

Следует отметить один важный момент: ресурсы, отправленные с помощью push, отсылаются только в ответ на первоначальный запрос. Загрузка ресурсов с помощью HTTP/2 push не осуществляется только на основании решения сервера, что клиент может нуждаться в них. Такие технологии, как WebSockets и server-sent events (SSE), допускают создание двустороннего потока, однако HTTP/2 не является двунаправленным; все действия инициируются запросом со стороны клиента. Ресурсы, отправленные с помощью push, представляют собой дополнения к ответу на первоначальный запрос. Когда первоначальный запрос завершается, закрывается и поток, и поэтому отправка других ресурсов невозможна, пока клиент не совершит еще один запрос. Таким образом, HTTP/2 push не является полноценной заменой WebSockets или SSE в том виде, в каком он определен в настоящее время, хотя, возможно, он станет таковым после внедрения некоторых расширений (см. раздел 5.9).

5.2 Как отправлять push-сообщения

Способ отправки зависит от используемого вами веб-сервера, так как на момент написания данной книги не все серверы поддерживают HTTP/2 push. Некоторые веб-серверы могут отправлять push-сообщения, в которых задаются HTTP-заголовок `link` или настройки конфигурации. Другие веб-серверы (например, IIS) требуют для этого написания отдельного кода, поэтому они могут отправлять push-сообщения только с динамически сгенерированных страниц, а не со статических файлов HTML. В документации вашего веб-сервера содержится информация о том, поддерживает ли он HTTP/2 push, а также рекомендации по его использованию. Далее в этой главе используются в основном примеры Apache, nginx и NodeJS. Данные концепции применимы к большинству веб-серверов HTTP/2, даже если в реализации детали немного различаются. Если ваш веб-сервер не поддерживает HTTP/2 push, вы можете применить обходной путь: использовать сети доставки содержимого и уже с их помощью отправлять push-сообщения.

5.2.1 Отправка push-сообщений с помощью HTTP-заголовка ссылки

Многие веб-серверы (например, Apache, nginx и H2O) и некоторые CDN (например, Cloudflare и Fastly) уведомляют веб-сервер о необходимости использования push-технологии с помощью HTTP-заголовка `link`. Когда сервер видит данные заголовки, он отправляет указанные в нем ресурсы с помощью push. Для Apache вы можете добавить такой заголовок с помощью следующей конфигурации:

```
Header add Link "</assets/css/common.css>;as=style;rel=preload"
```

Для ngix вы можете использовать похожий синтаксис:

```
add_header Link ";as=style;rel=preload"
```

Заголовки push-ссылок часто заключаются в условные операторы, что позволяет применять push только для определенных маршрутов или типов файлов. Например, в Apache, для загрузки с помощью push таблицы стилей CSS только с файлами index.html, а не со всеми ресурсами сразу вы можете использовать следующую конфигурацию:

```
<FilesMatch "index.html">
    Header add Link ";as=style;rel=preload"
</FilesMatch>
```

Для других веб-серверов HTTP-заголовки добавляются аналогичным образом, хотя не каждый из них использует метод HTTP-заголовка ссылки для отправки ресурсов с помощью push. Если сервер все же поддерживает такой метод, при ответе клиенту он считывает данные заголовки, запрашивает соответствующий ресурс и отправляет его. Атрибут `rel=preload` позволяет указать серверу на необходимый для отправки ресурс, а часть конфигурации `as=style` (которая указывает на тип ресурса) может быть необязательна. Атрибут `as` может использоваться для расстановки приоритета, однако он требуется не для всех серверов: например, Apache расставляет приоритет с помощью определения типа содержимого.

HTTP-заголовок предварительной загрузки и HTTP/2 push

Заголовки ссылок предварительной загрузки использовались еще до создания HTTP/2 и изначально задуманы как подсказка для клиента (см. главу 6). Данный заголовок позволяет браузерам получать ресурсы, не дожидаясь загрузки, прочтения и анализа всей страницы, благодаря чему у них есть время на то, чтобы решить, необходимо ли загружать эти ресурсы. Заголовки предварительной загрузки позволяют владельцу веб-сайта сказать: «Этот ресурс обязательно понадобится, поэтому я предлагаю вам запросить его как можно скорее, если он еще не у вас в кеше».

Во многих реализациях HTTP/2 заголовок ссылки предварительной загрузки получил иное предназначение с целью внедрения концепции push-загрузки, и поэтому данная подсказка вышла на несколько иной уровень. В случае, если вы собираетесь использовать предварительную загрузку, а не отправить ресурс с помощью push, вы можете использовать атрибут nopush:

```
Header add Link ";as=style;rel=preload;nopush"
```

На сегодняшний день не существует какого-либо стандартизированного способа, позволяющего осуществить обратный процесс (т. е. заявить, что мы хотим отправить заголовок ссылки с помощью push и на самом деле он не является заголовком предварительной загрузки). Однако веб-сервер H2O (и CDN Fastly, которую он использует[a]) добавил для таких случаев атрибут `x-http2-pushonly`:

```
link: ;as=script;rel=preload;x-http2-push-only
```

Заголовок предварительной загрузки также можно внедрить в сам HTML в теге HEAD следующим образом:

```
<link rel="preload" href="/assets/css/common.css" as="style">
```

Однако для HTTP/2 push обычно работает только использование HTTP-заголовка. Метод внедрения предварительной загрузки в HTML не подходит для HTTP/2 push, поскольку серверам сложно анализировать весь код и извлекать из него заголовки. Веб-браузерам же в любом случае приходится анализировать HTML, поэтому они поддерживают оба вышеописанных метода.

Для клиентских подсказок необходимо указывать атрибут `as`, однако при использовании HTTP/2 push он может и не требоваться. Во избежание путаницы я рекомендую указывать данный атрибут всегда. На вебсайте w3.org[b] вы можете найти полный набор атрибутов `as`, а также `script`, `style`, `font`, `image`, и `fetch`. Обратите внимание, что для некоторых из этих атрибутов (особенно для `font`) также требуется атрибут `crossorigin`[c].

Повторное использование заголовков предварительной загрузки для HTTP/2 push немного сбивает с толку. Многие говорят, что применение существующих функций для новых целей[d] – это плохая идея, и предлагают найти новые способы. Несмотря на это беспокойство, использование заголовков, похоже, растет. Еще одним преимуществом использования заголовка предварительной загрузки в качестве подсказки как для клиента, так и для push-сервера является то, что, в случае если и клиент, и сервер не поддерживают HTTP/2 push, они все же смогут использовать заголовок для предварительной загрузки ресурса с высоким приоритетом, получая таким образом прирост производительности. В разделе 5.8 я еще затрону тему директив предварительной загрузки и рассмотрю различные варианты их использования в HTTP/2 push. Вся эта сноска предназначена для тех читателей, которые понимают, что предварительная загрузка может быть подсказкой для клиентов.

---

[a] https://www.fastly.com/blog/optimizing-http2-server-push-fastly.
[b] https://www.w3.org/TR/preload/#as-attribute.
[c] https://drafts.csswg.org/css-fonts/#font-fetching-requirements.
[d] https://github.com/w3c/preload/issues/99.

При тестировании страниц в Apache вам следует отключить `PushDiary`, который будет пытаться предотвратить повторную отправку одних и тех же ресурсов по одному и тому же соединению (подробнее мы поговорим об этом в разделе 5.4.4):

```
H2PushDiarySize 0
```

Явный запрос обновления страницы в браузере (F5) заставит Apache игнорировать `PushDiary`, но во время тестирования его все же лучше отключить; в противном случае вы увидите противоречивые результаты. Для других серверов ситуация может обстоять подобным образом, и от-

слеживание push для них также следует отключать. Также вы можете отправить несколько заголовков, используя два ссылочных заголовка:

```
Header add Link "</assets/css/commoncss>;rel=preload;as=style"
Header add Link "</assets/js/common.js>;rel=preload;as=script"
```

или объединив их в один заголовок и разделив запятыми:

```
Header add Link " </assets/css/common.css>;rel=preload;as=style,
</assets/js/common.js>;rel=preload;as=script"
```

В главе 1 уже было сказано о том, что оба этих метода для HTTP синтаксически идентичны, поэтому вы можете использовать любой из них.

5.2.2 Просмотр ресурсов, отправленных с помощью HTTP/2 push

Ресурсы, загруженные с помощью push, отображаются на панели инструментов разработчика Chrome в столбце Initiator (инициатор обмена), как показано на рис. 5.6.

На нем вы можете увидеть, что второй ресурс (common.css) отправлен сервером. Также вы видите, что для данного запроса, в отличие от всех последующих, загрузка ресурса начинается сразу, и на каскадной диаграмме нет элемента `Waiting` (`TTFB`), отмеченного зеленым цветом.

Name	Status	Protocol	Scheme	Domain	Initiator	Size	Time
www.tunetheweb.com	200	h2	https	www.tunetheweb.com	Other	3.8 KB 6.1 KB	670 ms 666 ms
common.css /assets/css	200	h2	https	www.tunetheweb.com	Push / Other	5.6 KB 20.9 KB	183 ms 2 ms
LatoLatin-Thin.woff2 /assets/fonts	200	h2	https	www.tunetheweb.com	(index) Parser	42.5 KB 42.1 KB	387 ms 193 ms
googleanalytics.js /assets/js	200	h2	https	www.tunetheweb.com	(index) Parser	322 B 406 B	386 ms 384 ms
wrench-icon-48-202352.png /assets/images/wrench-icon	200	h2	https	www.tunetheweb.com	(index) Parser	1.6 KB 1.5 KB	386 ms 384 ms

Рис. 5.6 Ресурс, отправленный с помощью HTTP/2 push во вкладке Network (Сеть) панели инструментов разработчика Chrome.

На рис. 5.7 изображена загрузка той же страницы без push (запрос common.css переместился со второй позиции на третью и не был отправлен с помощью push).

Name	Status	Protocol	Scheme	Domain	Initiator	Size	Time
www.tunetheweb.com	200	h2	https	www.tunetheweb.com	Other	3.5 KB 6.1 KB	626 ms 621 ms
LatoLatin-Thin.woff2 /assets/fonts	200	h2	https	www.tunetheweb.com	(index) Parser	42.5 KB 42.1 KB	900 ms 267 ms
common.css /assets/css	200	h2	https	www.tunetheweb.com	(index) Parser	5.6 KB 20.9 KB	904 ms 901 ms
googleanalytics.js /assets/js	200	h2	https	www.tunetheweb.com	(index) Parser	322 B 406 B	905 ms 901 ms
wrench-icon-48-202352.png /assets/images/wrench-icon	200	h2	https	www.tunetheweb.com	(index) Parser	1.6 KB 1.5 KB	905 ms 901 ms

Рис. 5.7 Загрузка той же страницы, что и на рис. 5.6, без HTTP/2 push

```
[  0.017] recv (stream_id=13) :path: /assets/css/common.css
[  0.017] recv (stream_id=13) :method: GET
[  0.017] recv (stream_id=13) accept: */*
[  0.017] recv (stream_id=13) accept-encoding: gzip, deflate
[  0.017] recv (stream_id=13) user-agent: nghttp2/1.28.0
[  0.017] recv (stream_id=13) host: www.tunetheweb.com
[  0.017] recv PUSH_PROMISE frame <length=73, flags=0x04, stream_id=13>
          ; END_HEADERS
          (padlen=0, promised_stream_id=2)
```

Фрейм PUSH_PROMISE похож на фрейм HEADERS, отправляемый браузером для получения исходного ресурса, но все же имеет два важных отличия:

- данный фрейм отправляется сервером в браузер, а не наоборот. Он выступает предупреждением от сервера клиенту «Я собираюсь отправить вам ресурс»;
- он включает в себя promised_stream_id, который является идентификатором потока для отправленного ресурса, как показано в последней строке, и указывает, что ресурс будет отправлен с помощью push с идентификатором потока 2. Потокам, инициированным сервером (на данный момент только push-потоки), присваиваются четные номера.

После этого сервер возвращает первоначально запрошенный ресурс в потоке запроса (13), используя фрейм HEADERS, за которым следуют фреймы DATA. Затем он отправляет ресурс с помощью push в соответствующий поток (2), опять же используя фрейм HEADERS, за которым следуют фреймы DATA:

```
[  0.017] recv (stream_id=13) :status: 200
[  0.017] recv (stream_id=13) date: Sun, 04 Feb 2018 12:28:07 GMT
[  0.017] recv (stream_id=13) server: Apache
[  0.017] recv (stream_id=13) last-modified: Thu, 18 Jan 2018 21:52:14 GMT
[  0.017] recv (stream_id=13) accept-ranges: bytes
[  0.017] recv (stream_id=13) cache-control: max-age=10800, public
[  0.017] recv (stream_id=13) expires: Sun, 04 Feb 2018 15:28:07 GMT
[  0.017] recv (stream_id=13) vary: Accept-Encoding,User-Agent
[  0.017] recv (stream_id=13) content-encoding: gzip
[  0.017] recv (stream_id=13) link: ;rel=preload
[  0.017] recv (stream_id=13) content-length: 6755
[  0.017] recv (stream_id=13) content-type: text/html; charset=utf-8
[  0.017] recv (stream_id=13) push-policy: default
[  0.017] recv HEADERS frame <length=2035, flags=0x04, stream_id=13>
          ; END_HEADERS
          (padlen=0)
          ; First response header
[  0.017] recv DATA frame <length=1291, flags=0x00, stream_id=13>
[  0.017] recv DATA frame <length=1291, flags=0x00, stream_id=13>
[  0.018] recv DATA frame <length=1291, flags=0x00, stream_id=13>
[  0.018] recv DATA frame <length=1291, flags=0x00, stream_id=13>
[  0.018] recv DATA frame <length=1291, flags=0x00, stream_id=13>
[  0.018] recv DATA frame <length=300, flags=0x01, stream_id=13>
          ; END_STREAM
```

```
[  0.018] recv (stream_id=2) :status: 200
[  0.018] recv (stream_id=2) date: Sun, 04 Feb 2018 12:28:07 GMT
[  0.018] recv (stream_id=2) server: Apache
[  0.018] recv (stream_id=2) last-modified: Sun, 07 Jan 2018 14:57:44 GMT
[  0.018] recv (stream_id=2) accept-ranges: bytes
[  0.018] recv (stream_id=2) cache-control: max-age=10800, public
[  0.018] recv (stream_id=2) expires: Sun, 04 Feb 2018 15:28:07 GMT
[  0.018] recv (stream_id=2) vary: Accept-Encoding,User-Agent
[  0.018] recv (stream_id=2) content-encoding: gzip
[  0.018] recv (stream_id=2) content-length: 5723
[  0.018] recv (stream_id=2) content-type: text/css; charset=utf-8
[  0.018] recv HEADERS frame <length=63, flags=0x04, stream_id=2>
          ; END_HEADERS
          (padlen=0)
          ; First push response header
[  0.018] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.018] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.018] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.018] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.018] recv DATA frame <length=559, flags=0x01, stream_id=2>
          ; END_STREAM
```

5.2.3 Загрузка ресурсов посредством push из нисходящих систем с помощью заголовков ссылок

В случае, если для указания на необходимые ресурсы вы используете ссылочные заголовки, вам не нужно настраивать их в конфигурации веб-сервера. Как было сказано в главе 3, обычно – по соображениям производительности и безопасности – перед целевым приложением стоит веб-сервер, например Apache (это может быть сервер приложений, такой как Tomcat, NodeJS или обработчик PHP). Если данные серверы приложений проксируются через веб-сервер, поддерживающий HTTP/2 push с использованием заголовков ссылок (как Apache и ngix) при условии, что у вас есть возможность устанавливать заголовки ответов, сервер приложений может отправлять запросы на отправку ресурсов от веб-сервера, как показано на рис. 5.9.

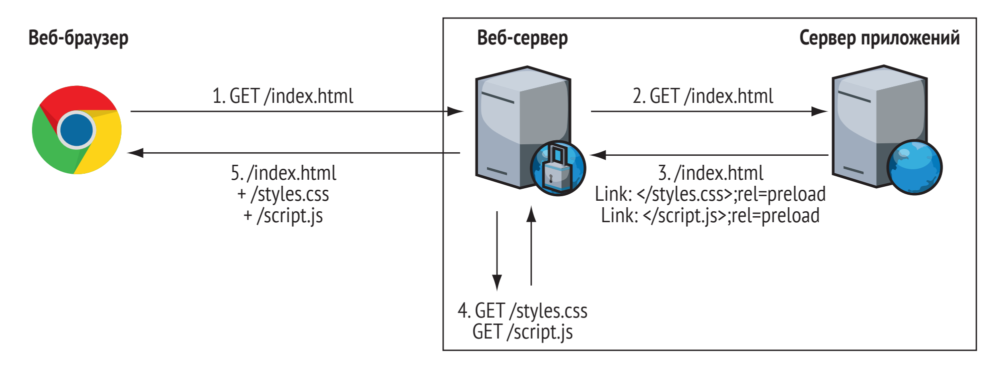


Рис. 5.9 Отправка заголовков ссылок HTTP/2 push с серверов целевых приложений

Использование ссылочных заголовков HTTP позволяет приложению сообщать веб-серверу, что нужно отправить, поэтому вся логическая часть может быть в одном месте, а каждый раз менять конфигурацию веб-сервера и код приложения необходимости нет. Данный процесс работает, даже если серверные соединения используют HTTP/1. В использовании HTTP/2 на внутренних соединениях нет необходимости, даже если вы хотите отправить ресурсы именно оттуда. Учитывая все возможные сложности (упомянутые в главе 3), это настоящий подарок судьбы! На рис. 5.10 изображено, как выглядит такой поток на диаграмме запросов и ответов.

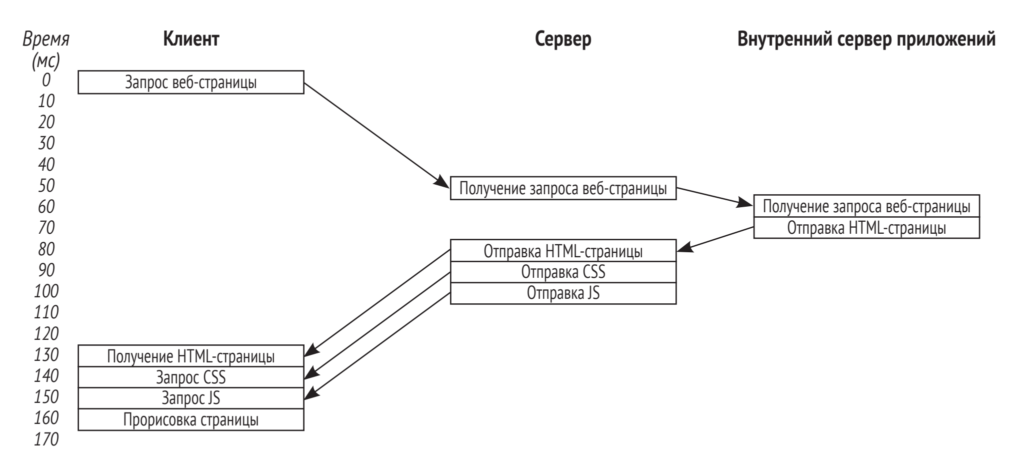


Рис. 5.10 **Использование ссылочных заголовков для push-загрузки ресурсов с внутреннего сервера приложений**

Для того чтобы увидеть пример подобного потока, создайте простой сервер с поддержкой HTTP/1.1, как показано в следующем листинге:

Листинг 5.1 Код службы HTTP/1.1 с ссылочным заголовком HTTP

```
var http = require('http')
const port = 3000

const requestHandler = (request, response) => {
  console.log(request.url)
  response.setHeader('Link','</assets/css/common.css>;rel=preload');
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write('<!DOCTYPE html>\n')
  response.write('<html>\n')
  response.write('<head>\n')
  response.write('<link rel="stylesheet" type="text/css"
href="/assets/css/common.css">\n')
  response.write('</head>\n')
  response.write('<body>\n')
  response.write('<h1>Test</h1>\n')
  response.write('</body>\n')
  response.write('</html>\n')
```

```
  response.end();
}

var server = http.createServer(requestHandler)
server.listen(port)
console.log('Server is listening on ' + port)
```

Поместите этот код в файл с именем app.js, а затем запустите его с помощью следующей команды:

```
node app.js
```

Вы должны увидеть строку с таким содержанием:

```
Server is listening on 3000
```

Данный код прослушивает порт 3000 и отправляет в ответ простую жестко вписанную в код веб-страницу, которая ссылается на таблицу стилей в теге `HEAD` и включает ее в заголовок ссылки. Для того чтобы проверить результат в другом окне, вы можете использовать curl:

```
$ curl -v http://localhost:3000
* Rebuilt URL to: http://localhost:3000/
* Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 3000 (#0)
> GET / HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.56.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Link: </assets/css/common.css>;rel=preload;as=style
< Content-Type: text/html
< Date: Sun, 04 Feb 2018 15:46:12 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
<
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" type="text/css" media="all"
    href="/assets/css/common.css">
</head>
<body>
<h1>Test</h1>
</body>
</html>
* Connection #0 to host localhost left intact
```

Для того чтобы разрешить вызов этого сервера через Apache, добавьте в конфигурацию Apache следующую строку. Необходимо включить `mod_proxy` и `mod_proxy_http`:

```
ProxyPass /testnodeservice/ http://localhost:3000/
```

Затем через Apache код вызовет необходимую службу, прослушивая HTTPS через порт 443. Этот код позволяет вам вызывать службу через HTTP/2 в браузере, даже если приложение NodeJS не настроено для поддержки HTTP/2 или HTTPS; Apache позаботится об этом за вас. На рис. 5.11 вы можете увидеть, как Apache отправляет через push таблицу стилей. В данном примере загруженный с помощью push ресурс (common.css) обслуживается Apache. Связанный ресурс может обслуживаться самим Apache, целевым приложением (в данном случае NodeJS) или другой целевой системой. Пока Apache может запрашивать ресурс, он может и отправлять любые такие ресурсы.

Name	Status	Protocol	Scheme	Domain	Initiator	Size
testnodeservice/ www.tunetheweb.com	200	h2	https	www.tunetheweb.com	Other	473 157
common.css /assets/css	200	h2	https	www.tunetheweb.com	Push / Other	7.2 K 20.9 K

Рис. 5.11 Ресурсы, указанные в ссылочном заголовке, могут допускать отправку push-сообщений из нижестоящих систем

Веб-сервер не может отправить ресурс для другого домена. Если вы загружаете страницу example.com, которая нуждается в изображениях, например с google.com, вы не сможете загрузить их с помощью push; сделать это может только google.com. В разделе 5.5.1 мы обсудим эту тему подробнее.

Предыдущий пример, где таблица стилей всегда загружается с помощью push, довольно прост, однако вы можете создавать и более сложные примеры на любом языке, используемом в нижестоящем сервере, проксируемом через веб-сервер (или CND) с использованием ссылочных заголовков HTTP. Приложение может само решать, что и когда отправлять, на основе того, что ему известно о запросе или пользовательском сеансе, но при этом оно переносит функцию фактической отправки на веб-сервер.

5.2.4 Предварительная push-загрузка ресурсов

Помимо использования ссылочных HTTP-заголовков, вы можете загружать ресурсы с помощью push и другими способами. Выбор способа зависит от вашего веб-сервера, так как процесс определяется конкретной реализацией. К примеру, Apache использует директиву `H2PushResource`:

```
H2PushResource add /assets/css/common.css
```

nginx предлагает схожий синтаксис:

```
http2_push /assets/css/common.css;
```

Неиспользованные push-ресурсы не отображаются в инструментах разработчика Chrome

Push-ресурсы, используемые на странице, отображаются во вкладке **Network** панели инструментов разработчика Chrome. Предзагрузчик информирует пользователя и дает ему подсказки касаемо предварительной загрузки. Благодаря этому отображаются все ресурсы, загруженные с помощью метода ссылочных заголовков (при условии, что включен атрибут as). Однако, если вы пользуетесь другим методом, а страница не использует загруженный ресурс, ресурсы будут загружаться в фоновом режиме и поэтому не отображаются на панели инструментов разработчика Chrome.

Если какой-либо push-ресурс не отображается в инструментах разработчика, проверьте, действительно ли он нужен странице. Если он не является обязательным, то его нет смысла загружать.

Метод прямой push-загрузки лучше, нежели загрузка с помощью заголовков HTTP. Его преимущество заключается в том, что серверу не нужно дожидаться заголовков, вместо этого он может сразу начать загрузку ресурса. Зависимые ресурсы могут начать загружаться уже во время обработки сервером исходного ресурса. Для простых статических ресурсов, генерация которых происходит довольно быстро (так как они извлекаются непосредственно с диска), вышеописанное преимущество может не иметь столь большого значения, как для ресурсов, которые генерируются медленнее[1]. На рис. 5.12 изображена диаграмма «запроса–ответа», аналогичная той, что была на рис. 5.5 в этой же главе. Однако здесь веб-страница генерируется за 100 мс. Возможно, это происходит из-за того, что для загрузки требуется поиск в базе данных или другая динамическая обработка.

На данном рисунке отмечен большой промежуток времени, в течение которого через соединение HTTP/2 ни отправка, ни прием ресурсов не происходит. Подобная задержка – это очень расточительно. Более того, все это напоминает проблемы блокировки заголовка строки, которые протокол HTTP/2 был призван решить. CSS и JavaScript обычно генерируются быстрее, поскольку они могут быть статическими, и сервер извлекает их с локального диска (или даже из собственного кеша). Освободившееся время может быть отведено для push-загрузки некоторых ресурсов, чтобы к моменту завершения генерации страницы зависимый ресурс уже был загружен (см. рис. 5.13).

---

[1] https://icing.github.io/mod_h2/earlier.html.

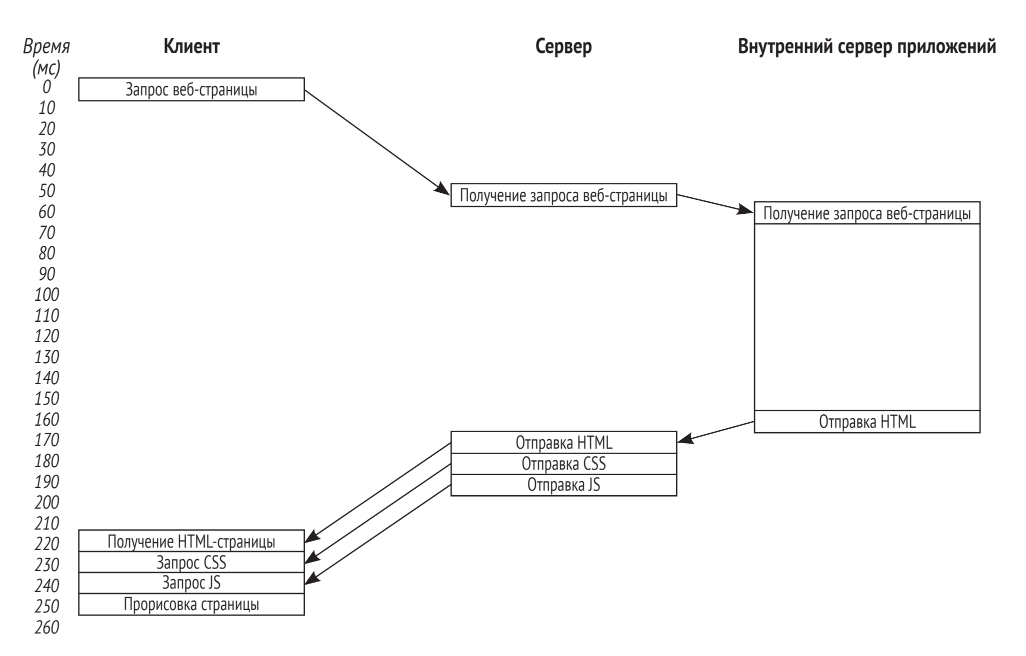


Рис. 5.12 Загрузка веб-страницы с учетом времени, затраченного на внутреннюю обработку

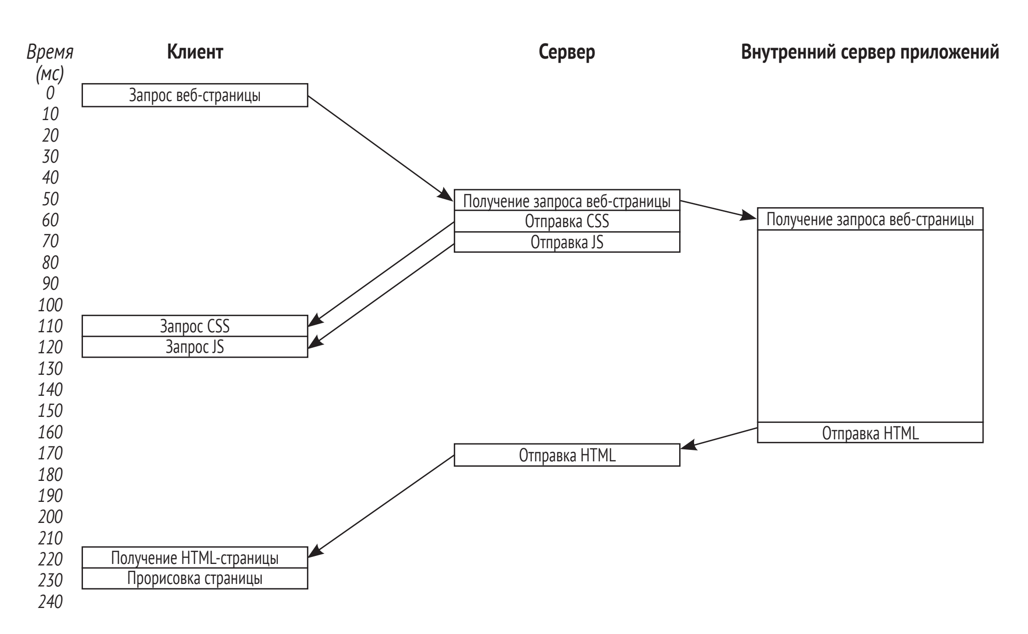


Рис. 5.13 Заблаговременная push-загрузка, позволяющая использовать время эффективнее

Листинг ниже демонстрирует имитацию задержки в NodeJS. Обратите внимание, что код `async/await` поддерживается только версией NodeJS 7.10 или новее.

Листинг 5.2 Код Node с ссылочными заголовками HTTP, задержка 10 мс

```
var http = require('http')
const port = 3000

async function requestHandler (request, response) {

  console.log(request.url)

  //Начало готовности ответа.
  response.setHeader('Link','</assets/css/common.css>;rel=preload ')

  //Пауза на 10 с для имитации медленного ресурса.
  await sleep(10000)

  //А теперь отправляем ресурс.
  response.writeHead(200, {"Content-Type": "text/html"})
  response.write('<!DOCTYPE html>\n')
  response.write('<html>\n')
  response.write('<head>\n')
  response.write('<link rel="stylesheet" type="text/css"
media="all" href="/assets/css/common.css">\n')
  response.write('</head>\n')
  response.write('<body>\n')
  response.write('<h1>Test</h1>\n')
  response.write('</body>\n')
  response.write('</html>\n')
  response.end();
}

function sleep(ms){
    return new Promise(resolve=>{
        setTimeout(resolve,ms)
    })
}

var server = http.createServer(requestHandler)
server.listen(port)
console.log('Server is listening on ' + port)
```

При обращении к этому коду с помощью nghttp и последующем перенаправлении ответа в grep с целью выделения только строк фрейма `recv` после установки соединения вы увидите 10-секундную задержку, необходимую для отправки фрейма `PUSH-PROMISE` (соответствует 10-секундному «засыпанию» в предыдущем коде):

```
$ nghttp -anv https://www.tunetheweb.com/testnodeservice/ | grep "recv.*frame"
[  0.209] recv SETTINGS frame <length=6, flags=0x00, stream_id=0>
[  0.209] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
[  0.213] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
[ 10.225] recv PUSH_PROMISE frame <length=73, flags=0x04, stream_id=13>
[ 10.225] recv HEADERS frame <length=647, flags=0x04, stream_id=13>
[ 10.225] recv DATA frame <length=139, flags=0x01, stream_id=13>
[ 10.226] recv HEADERS frame <length=108, flags=0x04, stream_id=2>
[ 10.226] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[ 10.226] recv DATA frame <length=1291, flags=0x00, stream_id=2>
```

```
[ 10.226] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[ 10.226] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[ 10.226] recv DATA frame <length=559, flags=0x01, stream_id=2>
```

В Apache, если вместо конфигурации, ожидающей ссылочные заголовки, выбрать конфигурацию push-загрузки с помощью `H2PushResource`, ресурсы будут загружены мгновенно, а 10-секундной задержки не возникнет, поскольку основной ресурс не будет препятствовать загрузке остальных:

```
$ nghttp -anv https://www.tunetheweb.com/testnodeservice/ | grep "recv.*frame"
[  0.248] recv SETTINGS frame <length=6, flags=0x00, stream_id=0>
[  0.248] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
[  0.253] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
[  0.253] recv PUSH_PROMISE frame <length=73, flags=0x04, stream_id=13>
[  0.253] recv HEADERS frame <length=675, flags=0x04, stream_id=2>
[  0.253] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.253] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.253] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.253] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.253] recv DATA frame <length=559, flags=0x01, stream_id=2>
[ 10.262] recv HEADERS frame <length=60, flags=0x04, stream_id=13>
[ 10.262] recv DATA frame <length=139, flags=0x01, stream_id=13>
```

Данное улучшение чрезвычайно полезно. В идеале при загрузке большинства ресурсов 10-секундной задержки происходить и не должно (она добавлена здесь для наглядности). Однако в реальности чем раньше вы приступите к push-загрузке, тем более эффективно вы сможете использовать доступную пропускную способность. Кроме того, загрузка других ресурсов не будет конфликтовать с основным запросом, так как он все равно будет готов позже.

Однако у данного способа есть и оборотная сторона. Одним из недостатков является то, что инициация push-уведомлений зависит уже в большей степени от приложения, а не от пользователя. Код состояния HTTP –`103 Early Hints`2–1 призван решить данную проблему. Он позволяет указывать требования к ресурсу заранее с помощью *предварительно загруженных ссылочных заголовков HTTP*. Как и все коды состояния в диапазоне `100`, данный код носит информационный формат, и приложение может проигнорировать его. Он позволяет получить ответ заранее и отправить при этом только заголовки (включая заголовки ссылок, необходимых для HTTP 2 push), за которыми последует стандартный код ответа `200`.

В мире HTTP/1.1 данный код выглядит как набор ответов, следующих друг за другом:

```
HTTP/1.1 103 Early Hints
Link: </assets/css/common.css>;rel=preload;as=style

HTTP/1.1 200 OK
```

[1] https://tools.ietf.org/html/rfc8297.

```
Content-Type: text/html
Link: </assets/css/common.css>;rel=preload;as=style

<!DOCTYPE html>
<html>
...и т. д.
```

На рис. 5.14 представлена диаграмма запросов и ответов.

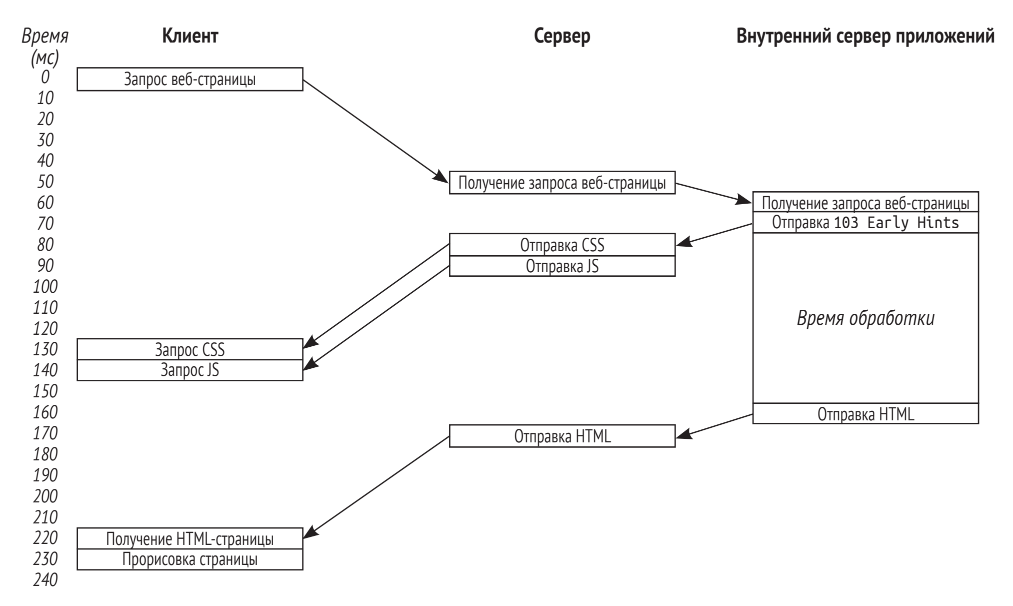


Рис. 5.14 Код состояния `103 Early Hints` сообщает веб-серверу о необходимости предварительной push-загрузки ресурсов

На этой диаграмме сервер отправляет предварительный ответ `103`, который говорит о том, что странице необходимы CSS и JavaScript. С помощью HTTP/2 push сервер отправляет клиенту данные статические ресурсы. Это происходит во время ожидания завершения генерации самой страницы. После завершения генерации и отправки страницы клиенту последний сразу же может воспользоваться push-ресурсами. Страница отобразится сразу после того, как клиент получит ее, однако есть ограничения: push-загрузка не распределяется между сервером приложений и веб-сервером, а также не должны использоваться приемы встраивания данных.

Вышеописанный процесс может работать медленнее, чем при использовании веб-сервера, поскольку в этом случае необходимо решить, какие ресурсы будут загружены с помощью push. На рис. 5.14 push-загрузка может начаться с отметки 60 мс, однако, если процесс осуществляется через ссылочные заголовки, она начнется только после отметки 80 мс, но выгода от использования для push-загрузки сервера приложений во многих сценариях перевешивает недостатки.

При использовании nghttp подобный сценарий выглядел бы следующим образом:

```
$ nghttp -anv https://www.tunetheweb.com/testnodeservice/ | grep
    "recv.*frame"
[  0.307] recv SETTINGS frame <length=6, flags=0x00, stream_id=0>
[  0.307] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
[  0.307] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
[  0.308] recv HEADERS frame <length=60, flags=0x04, stream_id=13>
[  0.308] recv PUSH_PROMISE frame <length=73, flags=0x04, stream_id=13>
[  0.309] recv HEADERS frame <length=675, flags=0x04, stream_id=2>
[  0.309] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.310] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.310] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.310] recv DATA frame <length=1291, flags=0x00, stream_id=2>
[  0.310] recv DATA frame <length=559, flags=0x01, stream_id=2>
[ 10.317] recv HEADERS frame <length=60, flags=0x04, stream_id=13>
[ 10.317] recv DATA frame <length=1291, flags=0x01, stream_id=13>
[ 10.317] recv DATA frame <length=1291, flags=0x00, stream_id=13>
[ 10.318] recv DATA frame <length=1291, flags=0x00, stream_id=13>
[ 10.318] recv DATA frame <length=1291, flags=0x00, stream_id=13>
[ 10.318] recv DATA frame <length=1291, flags=0x00, stream_id=13>
[ 10.318] recv DATA frame <length=300, flags=0x01, stream_id=13>
```

После завершения начальной настройки вы увидите следующее:

- на отметке 0,308 с во фрейме HEADERS в потоке 13 приходит ответ 103;
- фрейм PUSH-PROMISE (также в потоке 13) предупреждает клиента о предстоящем процессе push-загрузки;
- ресурс помещается во фрейм HEADERS и несколько последующих фреймов DATA, а затем на отметке 0,309 и 0,310 с все эти фреймы отправляются в потоке 2;
- после обработки реального ответа через 10,217 с он отправляется обратно в виде фрейма HEADERS, за которым следуют один или несколько фреймов DATA.

На момент написания этой книги поддержка нового кода ответа 103 Early Hints ограничена. Например, Node по умолчанию не поддерживает его[1], однако вы можете включить эту функцию с помощью сторонней библиотеки[2] или путем внесения в поток необработанного HTTP (что и делает библиотека). Apache может обрабатывать ответы с кодом 103. Он будет обрабатывать любые необходимые ссылочные заголовки. Однако намеренно пересылать ответ 103 браузеру он не станет, так как некоторые браузеры не поддерживают такие ответы, и могут возникнуть ошибки. Пересылку можно включить с помощью директивы H2EarlyHints53.

Поддержка ограничена еще и потому, что она предполагает отправку нескольких ответов на один запрос. Для ответов в диапазоне 100 такой исход допустим, но для других ответов он не совсем подходит, так как является дополнительным ответом к основному. Не все реализации HTTP

---

[1] Код состояния добавлен, но в данный момент использовать его возможности нет: https://github.com/nodejs/node/pull/16644.

[2] https://www.npmjs.com/package/early-hints.

[3] https://httpd.apache.org/docs/2.4/mod/mod_http2.html#h2earlyhints.

могут обработать такой ответ, ведь обычно они ожидают только один ответ. Другие коды состояния в диапазоне `100` (например, `100 Continue`, `101 Switching Protocols` и `102 Processing`) используются только для определенных сценариев, таких как переключение на WebSockets. Многие инструменты и библиотеки поддерживают разные коды состояния, даже те, которых не существовало на момент создания этих инструментов, однако лишь немногие средства могут правильно обрабатывать два запроса для установки кодов ответа вручную, как того требует код ответа `103`. Со временем поддержка появится, и это неизбежно. Таким образом, в будущем данный код ответа будет чрезвычайно полезен, и я надеюсь, что его будут поддерживать повсеместно.

5.2.5 Другие способы push-загрузки

Вы можете запустить push-загрузку не только с помощью веб-серверов; некоторые внутренние серверы приложений также позволяют разработчикам запускать push программным путем. В листинге 5.3 представлен способ создания простого сервера NodeJS с поддержкой push. Для этого вам потребуется модуль http2 и версия NodeJS v9 или новее.

Листинг 5.3 Служба Node с поддержкой push

```
'use strict'

const fs = require('fs')
const http2 = require('http2')

const PORT=8443

//Создание сервера HTTP/2 с сертификатом HTTPS и ключом.
const server = http2.createSecureServer({
  cert: fs.readFileSync('server.crt'),
  key: fs.readFileSync('server.key')
 })

//Обработка всех входящих потоков.
server.on('stream', (stream, headers) => {

  //Проверка, поддерживает ли входящий поток push на уровне соединения.
  if (stream.session.remoteSettings.enablePush) {

      //Если он поддерживает push, отправляем файл CSS.
      console.log('Push enabled. Pushing CSS file')

      //Открываем файл для чтения.
      const cssFile = fs.openSync('/www/htdocs/assets/css/common.css', 'r')

      //Получаем параметры файла для включения в HTTP-заголовок ответа.
      const cssStat = fs.fstatSync(cssFile)
      const cssRespHeaders = {
         'content-length': cssStat.size,
         'last-modified': cssStat.mtime.toUTCString(),
         'content-type': 'text/css'
      }
```

```
      //Отправка Push Promise для файла
      stream.pushStream({ ':path': '/assets/css/common.css' },
      (err, pushStream, headers) => {
        //Отправка файла во вновь созданный pushStream
        pushStream.respondWithFD(cssFile, cssRespHeaders)
      })
  } else {
      //Если push не поддерживается, заносим это в лог.
      console.log('Push disabled.')
  }
  //Ответ на исходный запрос.
  stream.respond({
    'content-type': 'text/html',
    ':status': 200
  })
  stream.write('<DOCTYPE html><html><head>')
  stream.write('<link rel="stylesheet" type="text/css" media="all"
ref="/assets/css/common.css">')
  stream.write('</head><body><h1>Test</h1></body></html>')
})

//Запуск сервера для прослушивания назначенного порта.
server.listen(PORT)
console.log(`Server listening on ${PORT}`)

})
```

Данный код позволяет NodeJS отправлять ресурсы в ваш браузер с помощью push. В этом простом примере поддержка HTTP/2 осуществляется только с помощью HTTPS. Для настоящего сервера вам, вероятно, потребуется HTTP/1.1 и HTTP (вот почему перед сервером приложений должен стоять веб-сервер, такой как Node). Другие языки программирования (такие как ASP.NET и Java) имеют аналогичные способы передачи ресурсов с помощью push.

Поддержка push на всех уровнях

Как я уже упоминал ранее, точкой входа в систему является балансировщик нагрузки или веб-сервер (который часто называют пограничным). После этого осуществляются прокси-запросы к внутреннему серверу приложений или службе. На самом деле я рекомендую всегда ставить веб-серверы перед динамическим сервером приложений, поскольку они обеспечивают прирост производительности и, кроме того, гарантируют безопасность. Когда речь заходит о HTTP/2 push, многие думают, что для всей сетевой инфраструктуры предпочтительнее использовать HTTP/2, чтобы сделать возможной push-отправку и загрузку между элементами инфраструктуры. Однако зачастую при таком варианте использования возникают некоторые ошибки, особенно когда задействован посредник. Что же делать, если сервер приложений и пограничный сервер поддерживают push, а клиент – нет, или наоборот? Как отследить перемещение push-ресурсов между тремя (или более) сторонами?

Спецификация HTTP/2 гласит[a]:

> *Посредник может получать push-ресурсы с сервера, но не пересылать их клиенту. Иными словами, именно он решает, как использовать push-информацию. Точно так же посредник может отправить клиенту дополнительные ресурсы с помощью push, и в этом процессе не будет задействован сервер.*

На самом деле, гораздо проще разрешить обрабатывать push пограничному серверу и делать это с помощью заголовков HTTP-ссылок (с кодом `103 Early Hints` или без него). Иногда случается так, что сервер приложений передает веб-серверу информацию об отправке ресурса с помощью push, и это может выглядеть как некий окольный путь. Однако такой путь намного проще, поскольку он позволяет серверам приложений использовать push для ресурсов, находящихся вне их контроля (это, например, статические файлы и мультимедиа, которые хранятся на уровне веб-сервера).

На момент написания этой книги мне неизвестны веб-серверы, которые поддерживали бы HTTP/2 push на всех уровнях. Прокси-модуль HTTP/2 Apache (`mod_proxy_http2`) – это одна из немногих реализаций серверных HTTP/2-соединений. Он отключает использование push для внутренних соединений, а для предотвращения возникновения неполадок использует фрейм `SETTINGS`[b].

В главе 3 мы обсудили варианты настройки инфраструктуры для поддержки HTTP/2. Возвращаясь к этому, можно сказать, что еще одной причиной, по которой поддержка HTTP/2 во всех элементах инфраструктуры, может быть, не нужна или даже нецелесообразна, является то, что веб-серверы и прокси-серверы зачастую не поддерживают HTTP/2. Такой исход будет актуален вплоть до того момента, пока поддержка HTTP/2 не станет повсеместной и больше не будет причин для отказа от этой версии протокола на всех уровнях.

[a] https://httpwg.org/specs/rfc7540.html#PushResources.

[b] https://github.com/icing/mod_h2/issues/154.

5.3 Как работает HTTP/2 push в браузере

Независимо от того, как вы используете push на стороне сервера, браузер обрабатывает данный процесс по-своему. Браузер помещает push-ресурс не на веб-страницу, а в кеш. Сама же веб-страница обрабатывается как обычно. Когда странице требуется какой-то ресурс, она проверяет кеш и находит там нужный ресурс, вместо того чтобы запросить его у сервера.

Детали этого процесса индивидуальны для каждого браузера, и поэтому в спецификации HTTP/2 они не описаны подробно. Большинство браузеров создало специальный *кеш для HTTP/2 push*, который несколько отличается от обычного знакомого всем веб-разработчикам кеша HTTP.

На момент написания этой книги лучшей «документацией» является публикация в блоге Джейка Арчибальда (Jake Archibald) (из команды Google Chrome[1]) об экспериментах с HTTP/2 push, в которой он рассказывает, как браузеры взаимодействуют с данным сервисом. В этой публикации подробно освещаются аспекты работы HTTP/2 в теории и на практике, а также описываются некоторые неожиданные способы использования. Благодаря данной работе выявлены некоторые ошибки; некоторые из них на момент написания книги уже исправлены, а некоторые нет.

HTTP/2 push – это новая концепция, и поэтому, для того чтобы устранить проблемы в ее реализациях на стороне браузеров (и, вероятно, серверов), необходимо проделать большую работу. Я постараюсь рассказать об основных ошибках, однако впоследствии могут появиться и новые.

5.3.1 Как работает кеш push

Все ресурсы, полученные с помощью push, хранятся в отдельном блоке памяти (кеш HTTP/2 push). Браузер может запросить их из кеша и загрузить на страницу. Как правило, кешированные заголовки (при наличии) также сохраняются в HTTP-кеше браузера. Однако существуют и исключения из этого правила, например браузеры на базе Cromium (Chrome и Opera) не кешируют ресурсы для неподтвержденных сертификатов (в их число входят самоподписанные сертификаты; веб-страницы, где они установлены, отмечены значком красного замка). При обнаружении ошибки сертификата функция кеширования будет недоступна[2]. Вы можете использовать HTTP/2 push, только если страница отмечена знаком зеленого замка, для чего необходимо наличие проверенного сертификата. В случае, если у вас самоподписанный сертификат, его необходимо внести в хранилище доверенных сертификатов браузера; в противном случае отправка или получение ресурсов с помощью push будет невозможным[3].

Кеш push – это далеко не первый источник, к которому браузер обращается при поиске ресурсов. Вообще, данный процесс зависит от выбранного браузера, однако эксперименты показывают, что, если ресурс можно запросить из обычного *HTTP-кеша*, браузер не станет обращаться к кешу push. Даже если push-ресурс новее, чем тот, что находится в кеше, браузер все же предпочтет второй (если исходя из *заголовков управления кешем* он решит, что данный ресурс приемлем). Таким же образом проверяются *Service workers* (*SW*). Push-загрузка ресурсов, которые не будут использоваться впоследствии, нецелесообразна. На рис. 5.15 показан процесс загрузки ресурса, необходимого для страницы в браузере Chrome, а также кеши, к которым обращается браузер при загрузке ресурсов.

---

1 https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/.
2 https://bugs.chromium.org/p/chromium/issues/detail?id=103875#c8.
3 https://bugs.chromium.org/p/chromium/issues/detail?id=824988.

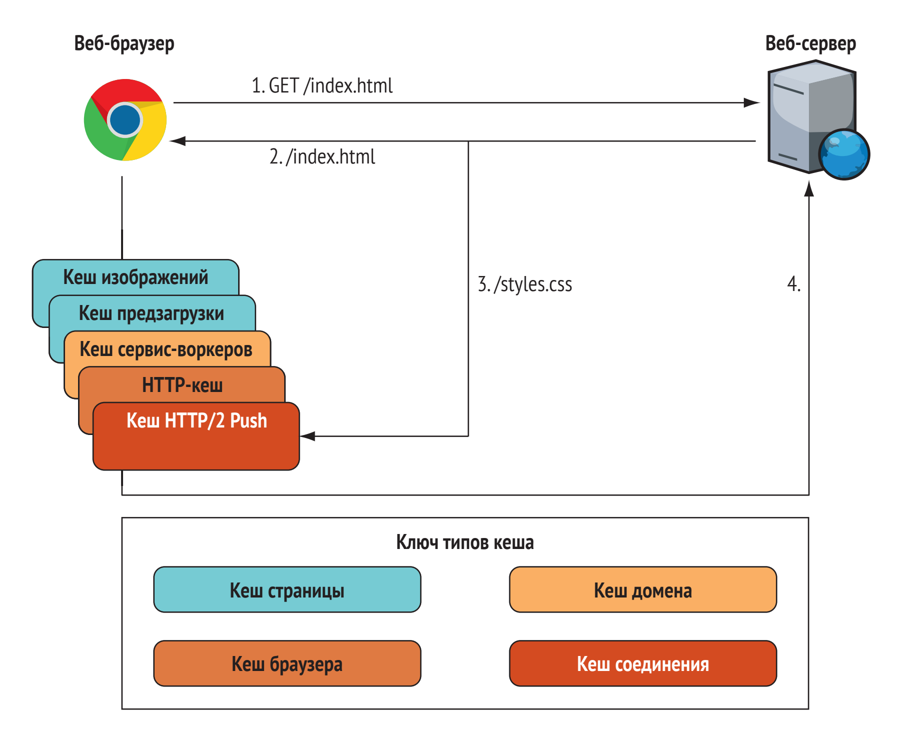


Рис. 5.15 Взаимодействие браузера с HTTP/2 push

При запросе к веб-серверу (1) и возвращении обратно в браузер (2) все push-ресурсы помещаются в push-кеш HTTP/2 (3). Перед отправкой запроса на веб-сервер кеши проверяются по порядку (4). Ниже приводится краткое описание каждого вида кеша:

- *кеш изображений* (*image cache*) – это временный кеш, который создается в памяти используемой страницы с целью предотвращения получения одного и того же изображения дважды. Такая ситуация может возникнуть, например, если на изображение ссылаются дважды. Когда пользователь закрывает страницу, кеш стирается;
- *кеш предварительной загрузки* (*preload cache*) – это также временный кеш в памяти страницы, в котором хранятся предварительно загруженные ресурсы (см. главу 6). Опять же, этот кеш полностью зависит от конкретной страницы, поскольку в предварительной загрузке других страниц заранее необходимости нет;
- *service workers* (*SW*) – это фоновые приложения, разработанные сравнительно недавно. Они работают независимо от веб-страницы и действуют как посредники между веб-страницей и веб-сайтом. Они позволяют веб-сайту функционировать как автономное приложение, даже при потере сетевого соединения. У SW есть собственные кеши, связанные с доменом;
- *HTTP-кеш* – это основной кеш, хорошо известный большинству разработчиков. Он является постоянным и располагается на диске. Од-

новременно его может использовать и браузер, но тогда его размер будет ограничен, чтобы хватило места для всех доменов;
- *кеш HTTP/2 push* – это временный кеш, хранящийся в памяти. Он зависит от соединения, и браузер обращается к нему в последнюю очередь.

Файл styles.css, отправленный сервером, помещается в push-кеш HTTP/2. Если браузеру необходим этот файл, перед отправкой сетевого запроса источнику он будет проверять все кеши по порядку, поскольку он не владеет информацией о том, отправил ли сервер данный ресурс. Если файл styles.css находится в основном кеше HTTP, браузер возьмет его оттуда, даже если в кеше HTTP/2 push есть более новая копия. Инструмент chrome://net-export, описанный в разделе 4.3.1, может предоставить вам сводку невостребованных push-ресурсов для всех текущих активных страниц, как показано на рис. 5.16.

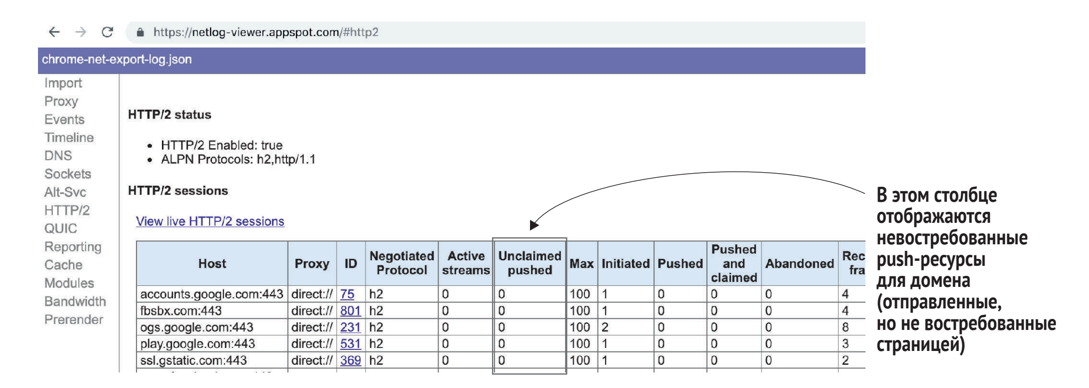


Рис. 5.16 Отслеживание невостребованных push-ресурсов в Chrome

Кеш HTTP/2 push зависит от соединения, поэтому если соединение уже не используется, то и push ресурсы в нем тоже. В HTTP-кеше, с которым привыкло работать большинство разработчиков, все происходит иначе. Например, в первом случае при потере соединения теряется и push-кеш, а также все загруженные ресурсы, которые не были использованы (что очень затратно). Если используется другое соединение, push-ресурсы из кеша предыдущего соединения не могут быть использованы. Однако HTTP/2 использует только одно соединение, поэтому такие ситуации не представляют серьезных проблем. Стоит помнить, что ситуации могут отличаться в зависимости от используемых браузеров, которые реализуют функции по-разному. Ранее я рассказывал о запросах без идентификаторов, для обработки которых большинство браузеров использует отдельное соединение. Однако сейчас *рабочая группа по технологиям веб-гипертекстовых приложений* (*WHATWG – Web Hypertext Application Technology Working Group*) разрабатывает изменения, которые позволят использовать одно соединение как для запросов с идентификаторами, так и без них[1]. Пока данная проблема не исправлена, вы не сможете от-

[1] https://github.com/whatwg/fetch/issues/341.

правлять шрифты из разных источников (загруженные из другого домена, включая сегментированные домены), так как такие запросы не содержат идентификаторов. Кроме того, отдельные вкладки или процессы в браузере могут инициировать новые соединении. Так происходит, например, в Edge, а Chrome и Firefox используют общее соединение между всеми вкладками. Что касается Safari, он открывает несколько соединений на одной вкладке[1]. Поскольку кеш HTTP/2 push работает на уровне соединения, а не на уровне страницы, можно отправлять ресурсы для будущей навигации по страницам, однако ввиду временного характера этого кеша в сочетании с тем, что возможна угроза прерывания соединения, это нецелесообразно.

Если ресурс был «востребован» браузером из push-кеша, в конечном итоге он удаляется и не может быть снова загружен, хотя, если заданы заголовки HTTP `cache-control`, его можно «востребовать» из HTTP-кеша браузера. Push-кеш также отличается от HTTP-кеша тем, что некешируемые ресурсы (те, которые установлены с HTTP-заголовками `cache-control`, `no-cache` и `no-store`) могут быть загружены и прочитаны из push-кеша. Таким образом, он не является кешем в традиционном понимании, а скорее представляет собой зал ожидания для запросов. Йоав Вайс (Yoav Weiss), архитектор веб-производительности, называет его «хранилищем невостребованных push-потоков»[2], но уточняет, что этот термин запомнить сложнее, чем «push-кеш».

5.3.2 Отказ от push с помощью RST_STREAM

Посредством отправки в push-поток фрейма `RST_STREAM` с кодом `CANCEL` или `REFUSED_STREAM` клиент может остановить push-загрузку ресурса. К использованию этого фрейма клиент может прибегнуть в случае, если у браузера уже есть данный ресурс или по какой-либо другой причине (например, если пользователь покидает страницу во время ее загрузки, и поэтому браузеру уже не нужно загружать этот ресурс).

Может показаться, что такой способ помогает избежать чрезмерного использования ненужных браузеру ресурсов. Однако его недостаток состоит в том, что на отправку данного фрейма обратно на сервер требуется некоторое время, в течение которого сервер отправляет и фреймы `HEADERS` и `DATA`, и получается, что браузер отбрасывает их. Фрейм `RST_STREAM` является сигналом обратной связи, и поэтому он является не столь агрессивным фактором, как разрыв соединения, который в HTTP/2 невозможно осуществить без прерывания других потоков в соединении. На момент получения и обработки сервером фрейма `RST_STREAM` ресурс уже мог быть загружен полностью.

Данный фрейм полезен только в том случае, если браузер принимает решение, что ему не требуется загружать ресурс именно с помощью push. Если данный ресурс есть в HTTP-кеше, то браузер, скорее всего,

---

1 https://bugs.webkit.org/show_bug.cgi?id=172639.
2 https://blog.yoav.ws/tale-of-four-caches/.

воспользуется фреймом RST_STREAM для того, чтобы остановить push-загрузку. Но что, если, например, изображение большого размера уже загружено с помощью push, однако на странице на него не ссылаются? К примеру, страницу обновили, и данное изображение уже не используется, а инструкция push-загрузки еще не была обновлена. В таком случае браузер загрузит весь объемный ресурс, однако использовать его не будет. В зависимости от того, какой браузер вы используете, вы можете даже не знать о том, что происходит подобный процесс, так как в некоторых случаях он может не отображаться во вкладке **Network** на панели инструментов разработчика.

Подводя итог изложенному выше, отметим следующее: фрейм RST_STREAM – это хороший способ остановить поток, и особенно поток push-загрузки. Однако полагаться на него как на способ управления push-загрузкой не стоит. Избыточная push-загрузка – это пустая трата ресурсов, и, даже если ваш сервер может с ней справиться, помните, что пропускная способность бывает ограничена. В частности, пропускную способность ограничивают, например, в мобильных соединениях. Таким образом, при отправке ненужных ресурсов, вы «платите» больше, не говоря уже о том, что потраченную впустую пропускную способность вы могли бы потратить на отправку *действительно* нужных ресурсов.

5.4 Условная push-загрузка

Push-загрузка ненужных ресурсов – это один из самых значительных рисков HTTP/2 push. Другие риски связаны с проблемами реализаций, о которых мы говорили ранее в этой главе (например, если соединение не используется повторно). Однако основная проблема заключается в push-загрузке ресурсов, которые уже есть в браузере.

Например, если вы загрузите таблицу стилей заранее с помощью push, время загрузки страницы для первого запроса сократится. Однако, если то же самое будет происходить каждый раз, когда посетитель совершает запросы на вашем сайте, получается, что ресурс, который уже есть у пользователя (при условии, что заголовки управления кешем гарантируют кеширование ресурса), будет загружаться снова и снова, что нецелесообразно.

Итак, фрейм RST_STREAM не позволяет нам приостанавливать push-загрузку файлов без излишних затрат ресурсов. Тогда какие методы мы можем использовать, чтобы гарантированно не загружать ненужные клиенту ресурсы?

5.4.1 Отслеживание push на стороне сервера

Сервер может отслеживать push-загрузку файлов в определенных клиентских соединениях. Способ отслеживания зависит от сервера, соединения или, возможно, от идентификатора сеанса. Например, если вы загружаете ресурс с помощью push, сервер отмечает, что это соедине-

ние или сеанс не должны загружать его снова, даже в случае соответствующих запросов. Так работает, например, Apache, и именно поэтому при тестировании сервера вам необходимо отключить настройку `H2PushDiarySize`. Данная функция может быть реализована в веб-приложении, а не в самом программном обеспечении веб-сервера, что обеспечит веб-разработчику больший контроль над соединением.

Недостатком является то, что сервер делает эмпирическое предположение относительно использования push. Например, после очистки кеша браузера некоторые ресурсы могут быть недоступны, но сервер все же не станет отправлять их. Кроме того, отслеживание push-ресурсов на перегруженных серверах может быть ограничено, а серверы с балансировкой нагрузки, если они не работают постоянно с одним и тем же клиентом, зачастую не обладают полной информацией о push-ресурсах.

В конечном счете данный процесс очень сложен. Более того, он является лишь грубой попыткой добавить отслеживание состояния к протоколу HTTP, который по умолчанию не предусматривает его. В HTTP/2 концепция состояния стала применима к некоторым частям протокола (например, сжатие заголовка HPACK и состояния потока, как обсуждается в главах 7 и 8). Таким образом, эта проблема может (и должна) быть решена на уровне протокола, что позволит оптимизировать его реализации. В разделе 5.4.4 я расскажу об одном из предложений, касающихся этого вопроса (дайджесты кеша), но сейчас стоит рассмотреть другие методы, которые в настоящее время возможно воплотить в жизнь.

5.4.2 Условные HTTP-запросы

HTTP-заголовки `if-modified-since` или `etag`, полученные от клиента, сообщают серверу о том, что нужная страница уже есть в кеше браузера, но срок сеанса уже истек. Если вы всегда загружаете CSS-файл с помощью push, то при наличии в запросе таких заголовков вы можете не делать этого, поскольку таблица стилей, вероятно, будет загружена из кеша (возможно, даже на более долгий срок, чем страница, которая ссылается на нее, как это часто бывает). Использование условных заголовков проще, чем отслеживание push-загрузки на стороне сервера. Однако недостатки двух этих процессов схожи. Например, при использовании условных заголовков сервер также делает однозначное обоснованное предположение о ресурсах на стороне клиента, а также использует тот же сценарий перехода на другую страницу, которая загружает таблицу стилей из кеша.

5.4.3 Push-загрузка с помощью куки-файлов

Еще одним способом является фиксация факта push-загрузки файла на стороне клиентов. Для этого можно использовать куки, а также `LocalStorage` или `SessionStorage`. Это работает следующим образом: когда вы загружаете ресурс с помощью push, вы устанавливаете действительный для данного сеанса или на момент push-загрузки ресурса (хранящегося в кеше довольно долго) файл куки (ресурсы, хранящиеся в кеше не столь

долгое время). Каждый запрос страницы должен быть проверен на наличие куки, и, если он есть, вы можете загрузить ресурс с помощью push. Данный способ может быть реализован в любом клиентском приложении или даже в конфигурации сервера. Рассмотрим на примере Apache:

```
#Проверка, сообщает ли куки о наличии загруженного css
#Если нет, устанавливаем переменную окружения для использования позже
SetEnvIf Cookie "cssloaded=1" cssIsloaded

#Если куки нет, и это файл html, отправляем файл css
#и формируем куки уровня сеанса, поэтому в следующий раз отправки не будет
<FilesMatch "index.html">
    Header add Link "</assets/css/common.css>;as=style;rel=preload"
     env=!cssIsloaded
    Header add Set-Cookie "cssloaded=1; Path=/; Secure; HttpOnly"
     env=!cssIsloaded
</FilesMatch>
```

Подобный код может быть реализован на любом языке.

Данный метод является усовершенствованной версией двух предыдущих. Процесс значительно упрощается, ввиду того что нет необходимости отслеживать ресурсы на стороне сервера, а также вы отслеживаете немного больше в зависимости от статуса браузера. Однако файлы куки и HTTP-кеш – это не одно и то же. Несмотря на то что можете установить одинаковое время истечения срока действия, файлы куки можно сбрасывать независимо (например, отключить их в браузере или запустить сеанс в режиме инкогнито), хотя то же самое можно сказать и о любом другом методе отслеживания ресурсов на стороне сервера.

На момент написания этой книги использование файлов куки является наилучшим способом отслеживания push-загрузки файлов и их состояния в кеше.

5.4.4 Дайджесты кеша

Дайджесты кеша позволяют[1] предоставлять браузеру информацию о содержимом его кеша. После установки соединения браузер отправляет фрейм `CACHE_DIGEST`, в котором перечислены все ресурсы, хранящиеся на данный момент в HTTP-кеше определенного домена (или других доменов, которые используют это соединение; см. главу 6). Сервер получает содержимое кеша в виде URL-адреса с заголовком `etag`, позволяющим ему управлять версиями URL. Использование дайджестов эффективнее вышеописанных «обходных» путей, в которых содержимое кеша приходилось «угадывать». Дайджесты позволяют браузеру предоставить серверу однозначную информацию о содержимом кеша. Сервер способен запоминать содержимое кеша и даже обновлять его по мере отправки дополнительных ресурсов. Браузер отправляет фрейм `CACHE_DIGEST` всего один раз в начале соединения (желательно после первого запроса).

[1] https://tools.ietf.org/html/draft-ietf-httpbis-cache-digest.

В кеше могут храниться большие объемы данных, поэтому вместо отправки полных URL-адресов еще и с заголовками `etag` клиенту предлагается зашифровать их в дайджесте с помощью *фильтра с кукушкой* (Cuckoo Filter[1]). Сейчас я не буду вдаваться в подробности данной концепции, но скажу, что она позволяет отправлять данные о содержимом кеша без возникновения риска конфликтов (например, ложные предположения о содержимом кеша).

На момент написания этой книги дайджесты не имеют официального стандарта, и многие браузеры не используют эту концепцию. Интересно, что некоторые серверы (такие как Apache, http2server и H2O) добавили поддержку текущего проекта стандарта (который возник из реализации H2O). На сегодняшний день такие реализации используются только для отслеживания запросов, отправленных сервером, поскольку браузеры не пользуются фреймом `CACHE_DIGEST` для инициализации серверного кеша. Таким образом, серверы могут отслеживать отправленные ресурсы, и у них нет необходимости прибегать к HTTP/2 push (даже если это рекомендуется). Безусловно, такая функция очень полезна, однако она могла бы быть еще полезнее, если бы позволяла инициализировать состояние кеша браузера, для чего и требуется фрейм `CACHE_DIGEST`. Как я уже упоминал в начале данного раздела, эту функцию Apache вы отключили при тестировании HTTP/2 push посредством следующей конфигурации:

```
H2PushDiarySize 0
```

В этой строке указывается, что максимальный размер push-журнала – 0, что означает, что его у вас попросту нет. Следовательно, вы можете воспользоваться push, даже если ресурсы уже были отправлены. Если вы протестируете страницу без установки значения 0, вы увидите, что иногда push-загрузка будет применяться, а иногда нет, и это может вас запутать. По окончании тестирования удалите данную конфигурацию и установите конфигурацию по умолчанию `H2PushDiarySize` (256 записей на соединение) или любую другую. Другие серверы могут поддерживать реализации, схожие с теми, что используются в Apache, поэтому я настоятельно рекомендую вам ознакомиться с документацией сервера.

Локальные обработчики заданий в вашем веб-приложении позволят реализовать дайджест кеша на стороне браузера, так как они способны перехватывать HTTP-запросы и вносить в них изменения. Существует несколько таких реализаций[2]. Фрейм `CACHE_DIGEST` не является стандартом и не реализован в браузерах или на серверах, вследствие чего использовать его нельзя. Обработчики заданий отправляют дайджест кеша в HTTP-заголовке или куки. Вы можете использовать эту концепцию на своем сервере для инициализации дайджеста кеша. Для этого ваше веб-приложение должно отправить такой заголовок «вручную» (с помощью обработчиков), а затем использовать его для инициализации серверной части. Протестировать данную функцию очень интересно, однако было

---

1 https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf.

2 https://www.npmjs.com/package/cache-digest-immutable.

бы гораздо лучше, если бы она была стандартизирована и работала бы на стороне браузера. К сожалению, в январе 2019 года рабочая группа HTTP объявила, что не планирует продолжать работу по стандартизации дайджестов кеша[1].

Безопасность работы с дайджестами требует отдельного обсуждения. Кеш браузера может содержать конфиденциальную информацию, например посещенные ранее URL-адреса, также (даже без использования куки) могут создаваться цифровые отпечатки пользователя и т. п. Вероятнее всего, серверу в любом случае будет предоставлен доступ к некоторым из этих данных (в какой-то момент запросы все же должны быть отправлены на сервер), однако аспект безопасности все же вызывает некоторое беспокойство. Согласно текущей концепции, браузеры не могут отправлять дайджесты кеша в режиме конфиденциальности или же если пользователь отказывается от файлов куки. Проблемы безопасности и конфиденциальности – это еще одна причина, по которой на сегодняшний день работа по стандартизации функции дайджеста кеша была приостановлена.

5.5 К каким ресурсам применим HTTP/2 push

Сейчас вы имеете четкое представление, что такое HTTP/2 push и как он работает на сторонах сервера и клиента. Однако, помимо этого, важно знать, к каким файлам он может быть применим.

5.5.1 К чему может быть применим push?

Согласно спецификации HTTP/2 push, существует ряд основных правил его использования[2]:

- клиент может отключить push посредством установки во фрейме `SETTINGS` значения `0` для параметра `SETTINGS_ENABLE_PUSH`. В таком случае сервер не будет отправлять фрейм `PUSH_PROMISE`;
- push-запросы должны иметь возможность быть кешированными (`GET`, `HEAD` и некоторые запросы `POST`);
- push-запросы должны быть безопасны (обычно `GET` или `HEAD`);
- push-запросы не должны включать в себя тело запроса (хотя обычно они содержат тело ответа);
- push-запросы могут быть отправлены только в домены соответствующего сервера;
- использовать push может только сервер;
- ресурсы могут быть отправлены с помощью push только в ответ на текущий запрос. Сервер не может инициировать push без запроса.

В реальности push-запросами могут быть только запросы `GET`. Вышеописанные правила касаются по большей части того, в каких случаях вы

1 https://lists.w3.org/Archives/Public/ietf-http-wg/2019JanMar/0033.html.
2 https://httpwg.org/specs/rfc7540.html#PushResources.

можете использовать push, однако гораздо важнее знать, в каких случаях вам *следует* его использовать.

Ограничение, согласно которому push-запросы могут быть отправлены только на домены соответствующего сервера, запрещает применять push к ресурсам, которые сервер не обслуживает (прямо или косвенно). Использование Bootstrap, загруженного с getbootstrap.com (или jQuery, размещенный на jquery.com) ограничивает использование вашим сервером push. Вы все же можете проксировать запросы через свой сервер, однако для этого вам нужно будет обновить все ссылки. На этом этапе вы можете организовать локальный хостинг страницы и устранить сложности с ее проксированием.

Signed HTTP Exchanges (официальное название – Web Packaging)[1] позволит вам обслуживать ресурсы из вашего домена ровно так же, как если бы они поступали из исходного домена, что позволит вам эффективно использовать push относительно других доменов. Однако данный механизм на момент написания этой книги находится на этапе разработки. Пока что он не используется в браузерах и серверах, но на него все же следует обратить внимание.

5.5.2 К чему должен быть применим push?

Ключевой вопрос, на который должны знать ответ владельцы веб-сайтов, желающие использовать HTTP/2 push, – к каким ресурсам может быть применим push, а к каким, что возможно даже более важно, нет. HTTP/2 push предназначен для оптимизации производительности, но он может и тормозить обмен, в случаях если вы бездумно используете push и тратите пропускную способность на push-загрузку ресурсов, которые не понадобятся клиенту.

В идеале применять push-загрузку следует только к критически важным для страницы ресурсам. Push-загрузка неиспользуемых ресурсов – это пустая трата пропускной способности. В число таких ресурсов входят, например, ненужные файлы (ресурсы, на которые нет ссылок), файлы, которые клиент не может использовать (например, изображения, формат которых не поддерживает браузер клиента), а также ресурсы, которые не следует использовать по решению самого клиента (например, изображения, которые отображаются только для определенных размеров экрана). Как я уже говорил, push-загрузку следует применять только к критически важным ресурсам. Хотя иногда возникает соблазн применить ее ко всем элементам, которые требует страница, однако это может замедлить получение критически важных ресурсов (в зависимости от приоритетов загрузки ресурсов, запрошенных клиентом).

Кроме того, необходимо учитывать наличие файла в кеше клиента (см. раздел 5.3). Загружать ресурсы с помощью push следует только в том случае, если высока вероятность, что данного ресурса еще нет в кеше.

HTTP/2 push должен применяться для того, чтобы использовать время простоя сети максимально эффективно. Следовательно, push-загрузка

[1] https://tools.ietf.org/html/draft-yasskin-http-origin-signed-responses.

всех необходимых странице ресурсов вряд ли повысит производительность, поскольку в таком случае вы будете игнорировать любые приоритеты, которые расставит браузер. Команда Chrome опубликовала подробный документ, содержащий информацию о том, к каким файлам применима push-загрузка[1]. Одна из основных рекомендаций в нем гласит, что необходимо загружать минимальное количество нужных ресурсов только «для заполнения времени простоя сети, и не более». Другое исследование[2] подтверждает использование push именно таким образом. Именно поэтому предварительная push-загрузка и код состояния `103` являются важными дополнениями к базовой стратегии push.

Короче говоря, *лучше загрузить с помощью push меньше ресурсов, чем больше*. Худшее, что может случиться, если ресурс не получится загрузить с помощью push, – это то, что его все равно нужно будет запросить, а условия для этого будут не совсем оптимальными. С другой стороны, злоупотребление push-загрузкой может привести к загрузке ненужных ресурсов, что будет очень затратно для клиента, сети и сервера, а также замедлит загрузку страницы. Однако даже при чрезмерной push-загрузке *HTTP/2 push не должен кардинально нарушить загрузку страницы*. Производительность может значительно снизиться, однако спустя какое-то время страница все же загрузится.

5.5.3 Автоматизация push-загрузки

Также необходимо выработать стратегию push-загрузки и того, какие именно ресурсы вы будете загружать. Должен ли это делать владелец веб-сайта или разработчик, или все же этот процесс можно автоматизировать? Jetty[3] – это механизм Java, с помощью которого вы можете автоматизировать push[4]. Он может отслеживать запросы с заголовком `Referer` (а также другие запросы, следующие за ними). Результаты отслеживания механизм применяет для опережающего создания предполагаемых push-ответов на вероятные аналогичные запросы от других клиентов. Безусловно, он значительно упрощает процесс выбора ресурсов, которые вы будете загружать с помощью push, однако все зависит от того, согласны ли вы с тем, что механизм предлагает, и подходит ли это для вашего веб-сайта. Таким образом, разработка стратегии довольно сложна, а процесс автоматизации push должен быть индивидуален для каждого сервера. Реализация Jetty довольно интересна, и в сочетании с некоторыми формами дайджестов кеша ее может быть достаточно для предотвращения нежелательной push-загрузки. В качестве альтернативы владельцам веб-сайтов может потребоваться прямой контроль за данным процессом, так как они должны знать свой сайт лучше его

---

1 https://docs.google.com/document/d/1K0NykTXBbbbTlv60t5MyJvXjqKGsCVNYHyLEXIxYMv0/, также доступно по адресу https://goo.gl/89RLGQ.

2 https://calendar.perfplanet.com/2016/http2-push-the-details/.

3 https://www.eclipse.org/jetty/.

4 https://www.eclipse.org/jetty/documentation/current/http2-configuring-push.html.

посетителей и иметь более ясное представление о том, что необходимо загружать с помощью push.

5.6 Решение проблем HTTP/2 push

Легче всего контролировать HTTP/2 push в столбце Initiator вкладки **Network** в инструментах разработчика Chrome (аналогичным образом это происходит и в других браузерах на основе Chromium, таких как Opera). Но что делать, если в этом столбце вы не видите соответствующий ресурс? Этому может быть несколько причин, рассмотрим самые распространенные:

- *точно ли вы используете HTTP/2?* Если нет, то push-загрузка будет невозможна. Для того чтобы убедиться, что вы используете нужную версию протокола, добавьте столбец Protocol. В главе 3 представлены советы по устранению неполадок, возникающих при переходе на HTTP/2;
- *поддерживает ли ваш сервер HTTP/2 push?* На момент написания этой книги некоторые серверы и CDN все еще не поддерживают его. Узнать, поддерживает ли клиент push, можно только благодаря фрейму `SETTINGS`. Однако увидеть, поддерживает ли его сервер, просто взглянув на фрейм `SETTINGS` с помощью nghttp или страницу net-export в Chrome, невозможно;
- *находится ли ваш сервер позади остальной инфраструктуры?* Если ваш сервер находится за балансировщиком нагрузки или другими элементами инфраструктуры, которые завершают соединение HTTP/2, существует вероятность что он не будет поддерживать HTTP/2 push, даже если его поддерживает ваш сервер. Даже при наличии поддержки HTTP/2 push ваш сервер не сможет передавать push-ресурсы, которые уже были обработаны вышеупомянутыми элементами инфраструктуры.

  Если перед внутренним сервером приложений (например, Node и Jetty) размещен Apache, то он не позволит ему загружать ресурсы с помощью push. Для того чтобы это исправить, внутренний сервер должен использовать заголовки HTTP-ссылок;
- *загружается ли файл именно с помощью push?* Для того чтобы просмотреть наличие фрейма `PUSH_PROMISE` и, соответственно, сам файл, воспользуйтесь nghttp. Таким образом, вы сможете выяснить, связана ли эта проблема с браузером;
- *нужны ли эти файлы странице?* Если для загрузки странице не нужны ресурсы, браузер не станет использовать их, а Chrome не отобразит их во вкладке **Network**. Для того чтобы просмотреть сводку невостребованных push-ресурсов, воспользуйтесь инструментом chrome net-export, как показано на рис. 5.16 в разделе 5.3.1.

  Если вы осуществляете push-загрузку с помощью заголовка HTTP-ссылки с атрибутом `rel=preload` и `as`, Chrome сочтет ресурсы необходимыми для страницы (благодаря подсказке предваритель-

ной загрузки) и отобразит их на вкладке **Network**. С одной стороны, это очень полезно, а с другой стороны, это, напротив, может нас запутать.

Один из способов отладки – удалить атрибут `as` (например, `as=style`) из заголовка ссылки. Chrome не будет использовать ресурсы в качестве подсказки для предварительной загрузки, но ваш веб-сервер все же должен их отправлять (поскольку атрибут `as` не является обязательным для push, в зависимости от реализации). Если ресурсы не отображаются на вкладке **Network**, но отображаются там при наличии атрибута `as`, это означает, что вы загружаете с помощью push ресурс, который странице не нужен;

- *правильно ли вы используете push для своего сервера?* Использование HTTP/2 push зависит от вашего сервера. Многие (но не все) серверы используют заголовки HTTP-ссылок, поэтому логично предположить, что для push-загрузки они будут использовать именно этот способ. Изучите документацию сервера или интерактивные руководства, чтобы узнать, как правильно использовать push на вашем сервере;
- *точно ли сервер решил загружать ресурс не с помощью push?* При определенных обстоятельствах push-загрузка ресурса может не произойти, если вы реализовали ее процесс с поддержкой кеширования (см. раздел 5.4) или каким-либо другим методом. Если после обновления страницы или перезагрузки браузера (или даже сервера) некоторое время файлы отображаются как push-ресурсы, проверьте, установили ли вы их отправку с помощью push. Если вы пользуетесь Apache, вы можете отключить функцию push-журнала, которая пытается предотвратить push-загрузку ресурсов, которые, по мнению сервера, у клиента уже есть, посредством присвоения `H2PushDiarySize` значения 0. Такой способ может быть полезен при отладке, когда вам необходимо, чтобы сервер загрузил один и тот же push-ресурс несколько раз;
- *существует ли файл, который вы хотите загрузить с помощью push?* Во время указания ресурса, к которому вы хотите применить push-загрузку, легко сделать опечатку, а если ресурса не существует, его невозможно будет отправить. В таком случае в журналах вашего веб-сервера появится код состояния `404` (Not Found). Если вы используете nghttp, такой код появится в полученном в ответ фрейме. Если вы пользуетесь методом заголовка HTTP-ссылки `rel=preload` с атрибутом `as`, код `404` отобразится во вкладке **Network** или запрос войдет в число отмененных[1];
- *возможно, вы осуществляете push-загрузку по другому соединению, отличному от ожидаемого браузером?* Как обсуждалось в разделе 5.2.1, HTTP/2 push напрямую связан с соединением. Если при push-загрузке вы пользуетесь одним соединением, а браузер ожидает другого, он не будет использовать загруженный ресурс. Такая

[1] https://bugs.chromium.org/p/chromium/issues/detail?id=811077.

проблема часто возникает при загрузке шрифтов, поскольку они должны загружаться по несертифицированным соединениям, однако некоторые проблемы и особенности браузера могут привести к тому, что они будут загружаться по другому соединению. Наилучшим способом увидеть эту ситуацию является просмотр подключения в WebPagetest;

- *возможно, вы используете самозаверенный или другой ненадежный сертификат?* Chrome игнорирует push-запросы для ненадежных сертификатов HTTPS (включая самозаверенные фиктивные сертификаты, созданные для локального хоста[1]). Для того чтобы избежать этого, вам необходимо добавить свой сертификат в хранилище доверенных сертификатов вашего компьютера, тогда для всех push-запросов будет отображаться значок зеленого замка. Кроме того, Chrome требует наличия у сертификата действительного *альтернативного имени субъекта (Subject Alternative Name – SAN)*. Во многих руководствах по созданию самозаверенных сертификатов рассказывается только о поле Subjects, которое на данный момент является устаревшим, и поэтому даже после добавления таких сертификатов в хранилище они не распознаются. Для того чтобы предотвратить эту проблему и получить возможность пользоваться HTTP/2 push, необходимо заменить их на сертификаты, содержащие и Subject, и SAN.

5.7 Влияние HTTP/2 push на производительность

На разных веб-сайтах степень влияния HTTP/2 push на производительность может быть различна, она зависит от времени, которое занимает цикл приема-передачи (время, необходимое на обслуживание ресурсов), а также от оптимизации веб-сайта. На сегодняшний день HTTP/2 push используют лишь немногие сайты, ввиду чего какая-либо значимая информация о его влиянии на производительность практически отсутствует.

Ключом к эффективному использованию HTTP/2 push является эффективное использование «промежутков» пропускной способности в то время, когда соединение не используется. Особое значение этот аспект имеет для страниц, процесс генерации которых на стороне сервера сложен и долог. Для статических страниц это значение несколько меньше. Однако для них также существует потенциальная возможность экономии ресурсов в циклах приема-передачи, поскольку из-за ограничений пропускной способности и обработки push-ресурсы, вероятнее всего, так или иначе будут стоять в очереди позади основных ресурсов с более высоким приоритетом, что приведет к задержке (половина цикла приема-передачи). Вы можете увидеть этот процесс на рис. 5.17. На обеих диаграммах первые четыре ресурса выглядят одинаково, поэтому от использования push здесь мало пользы.

---

1 https://bugs.chromium.org/p/chromium/issues/detail?id=824988.

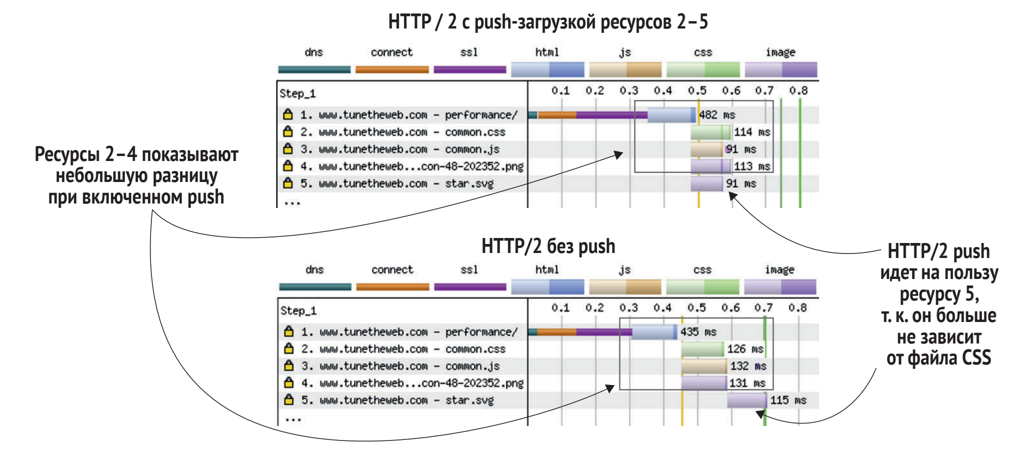


Рис. 5.17 Ресурсы, запрошенные с помощью HTTP/2 push и другие запрошенные ресурсы, поступают в разное время

Однако push-загрузка пятого ресурса уже оправданна. Таким образом, не нужно дожидаться загрузки таблицы стилей, чтобы проверить нужна ли она. (Позже в этой главе я расскажу о предварительной загрузке, которая позволяет получить аналогичные преимущества.) В данном примере push используется несколько нестандартно, так как благодаря этому исчезает необходимость встраивания ресурсов. Однако при правильном использовании вы все же сможете извлечь необходимые вам преимущества.

На 102-й встрече IETF в Монреале в июле 2018 года Akamai[1] и команда Chrome[2] представили свои наблюдения о влиянии HTTP/2 push на производительность. Akamai отметили значительные улучшения при применении push к критически важным ресурсам. Chrome также отметили небольшие улучшения, однако они проводили несколько другой эксперимент: они отключали push и анализировали разницу. Ввиду того, что исследования проводились разными способами, возникают некоторые вопросы. Являются ли клиенты Akamai более репрезентативными для интернета, ведь Chrome исследовал только те сайты, которые уже используют push (а их немного и, вероятно, их владельцы – сторонники HTTP/2)? Отвечает ли исследование Akamai на вопрос, к каким ресурсам необходимо применять push, и чье решение является более правильным – их или Google, где сайты выбирали эти ресурсы сами?

В ходе исследования были выделены две основные проблемы (особенно командой Chrome): HTTP/2 push использовали лишь немногие сайты (0,04 % сеансов HTTP/2, по данным Chrome), вследствие чего потенциально может возникнуть снижение производительности. Команда Chrome даже задалась вопросом, заметят ли пользователи отключение

1 https://github.com/httpwg/wg-materials/blob/gh-pages/ietf102/akamai-server-push.pdf.

2 https://github.com/httpwg/wg-materials/blob/gh-pages/ietf102/chrome_push.pdf.

push. Низкий уровень использования говорит о том, что реализация HTTP/2 push на серверах довольно сложна, ввиду чего многие люди сомневаются в его эффективности и предлагают альтернативны.

5.8 Push или предварительная загрузка?

Даже при должной работе (что далеко не всегда удается реализовать) HTTP/2 push имеет множество нюансов. При его использовании существуют явные риски, такие как снижение пропускной способности и скорости работы сайта (несмотря на то что HTTP/2 push призван ее ускорить). Я упоминал ранее, что сущность этих рисков заключается не в том, что владельцы сайтов приведут свои страницы в негодность, а в том, что они впустую потратят ресурсы, которые следовало бы использовать в другом месте. Одной из основных проблем является неосведомленность сервера о содержимом HTTP-кеша браузера. Возможно, что дайджесты кеша смогут решить эту проблему, если, конечно, для них разработают стандарт. Пока этого не произошло, многие люди задаются вопросом, готов ли HTTP/2 push к переходу в массовое использование, или все же пока что следует довольствоваться методом *предварительной загрузки*.

Предварительная загрузка[1] помогает заранее указать браузеру, что для загрузки страницы необходим определенный ресурс. Как упоминалось в разделе 5.1.1, вы можете реализовать предварительную загрузку посредством использования заголовка HTTP-ссылки (с атрибутом `nopush`, для того чтобы избежать push-загрузки ресурсов):

```
Link: "</assets/css/common.css>;rel=preload;as=style;nopush"
```

В HTML это выглядит следующим образом:

```
<link rel="preload" href="/assets/css/common.css" as="style">
```

Независимо от выбранного метода браузер должен воспринимать эту строку как индикатор того, что заявленный ресурс обладает высоким приоритетом. Как я уже говорил ранее, разница HTTP/2 push и предварительной загрузки заключается в том, что для последней особенно важен атрибут `as`; если его исключить, браузер или не сможет распознать необходимость предварительной загрузки, или нужный ресурс загрузится дважды.

В аспекте push-загрузки ресурса до его запроса предварительная загрузка уступает в скорости HTTP/2 push. Однако она является запросом, инициируемым браузером, и имеет также ряд преимуществ:

- браузер знает, что находится в его кеше, и на основе этой информации принимает решение о необходимости запроса того или иного ресурса. В отличие от HTTP/2 push при предварительной загрузке ресурсы, которые уже есть у клиента, не будут загружаться повторно. Если в кеше браузера уже имеется необходимый ресурс, браузер

[1] https://w3c.github.io/preload/.

проигнорирует запрос предварительной загрузки. Однако многие серверы HTTP/2 для push-загрузки ресурсов используют заголовки HTTP-ссылок, поэтому, если вы не хотите применять push-загрузку, вам следует добавить атрибут `nopush`;
- при использовании метода предварительной загрузки возникает меньше проблем и сложностей с push-кешем, поскольку в данном случае задействован только HTTP-кеш. Если предварительно загруженный ресурс не используется, вы все равно потратите время на его загрузку, однако это происходит независимо от того, используете ли вы предварительную загрузку, или HTTP/2 push;
- с помощью предварительной загрузки вы можете загружать ресурсы из других доменов, в то время как HTTP/2 push вы можете использовать только в собственном домене;
- на панели инструментов разработчика Chrome отображаются все предварительно загруженные ресурсы независимо от того, были они использованы или нет. Однако, что касается push-ресурсов, то отображаются только использованные при загрузке страницы (хотя есть и обходной путь: вы можете отправлять заголовок HTTP-ссылки предварительной загрузки для каждого push-ресурса).

На сегодняшний день ряда этих преимуществ может быть достаточно, и поэтому некоторые люди советуют пользоваться именно менее рискованным методом предварительной загрузки. Согласно анализу, проведенному Хуманом Бехешти (Hooman Beheshti) из Fastly, в феврале 2018 года HTTP/2 push использовали всего лишь 0,02 % сайтов[1] (и это через три года после официального выхода HTTP/2). Схожие результаты получила и команда Chrome, чей эксперимент описан в разделе 5.7. По ряду веских причин некоторые люди не решаются использовать эту технологию. Особенно если учесть, что существует метод предварительной загрузки, который обеспечивает те же преимущества, но со значительно меньшими рисками.

Предварительная загрузка с указанием кода состояния `103` еще больше приближает ее к HTTP/2 в плане прироста производительности. Данный код может быть отправлен вместе с заголовками HTTP-ссылок предварительной загрузки, что будет указывать браузеру на необходимость начала запроса ресурсов, для загрузки которых требуется относительно много времени. Таким образом, к нужному моменту, в зависимости от страницы и времени, которое требуется на ее генерацию, ресурсы уже могут быть доступны браузеру. В разделе 5.2.4 я рассказал, как код состояния `103` совместно с HTTP/2 push позволяет оптимизировать время обработки страницы. На рис. 5.18 представлена диаграмма, повторяющая уже имеющуюся в этой главе, поэтому, если вы не хотите возвращаться, можете просто взглянуть на нее.

На этом рисунке видно, что внутренний сервер приложений, для того чтобы сообщить веб-серверу о необходимости начала push-загрузки во время обработки запрошенной страницы, использует ответ `103 Early`

---

1 https://www.youtube.com/watch?v=wR1gF5Lhcq0.

Hints. Здесь данный ответ используется только для указания веб-серверу о необходимости push, и поэтому он может быть «проглочен» веб-сервером, так как отправлять его клиенту бесполезно (и, как говорилось ранее, некоторые браузеры не в состоянии должным образом обработать ответ 103).

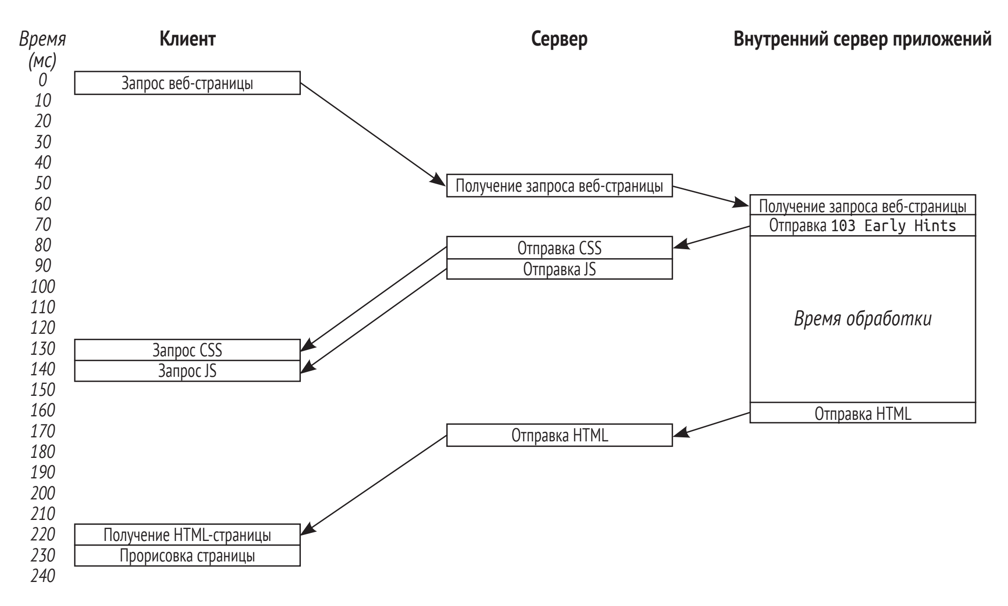


Рис. 5.18 103 Early Hints сообщает веб-серверу о необходимости начала push-загрузки заранее

Если такой вариант реализации push вас не устраивает, по всем причинам, обсуждаемым в этой главе, вы можете воспользоваться менее рискованным вариантом и отказаться от использования HTTP/2 push. Альтернативным вариантом является отправка браузеру ответа 103 (при условии, что браузер его поддерживает). Таким образом, браузер сможет использовать предварительно загруженные заголовки ссылок HTTP для предварительной загрузки ресурсов, как показано на рис. 5.19.

Конечно, этот процесс несколько медленнее, чем HTTP/2 push, поскольку браузеру все так же необходимо запрашивать ресурсы, однако теперь он может начать делать это заранее. В приведенном выше примере вы можете увидеть, что запрос дополнительных ответов накладывается на процесс отправки главной страницы. Такой «нахлест» несколько усложняет ситуацию, однако на практике все обычно происходит именно так. Преимущество предварительной загрузки заключается в том, что вы с меньшей вероятностью потеряете пропускную способность вашего соединения. Если в кеше браузера уже есть необходимые ресурсы, он не станет запрашивать их. В зависимости от времени предварительно загруженные ресурсы могут быть получены до того, как будет получена страница, для генерации которой они нужны. Таким образом, данный процесс становится таким же быстрым, как и использование HTTP/2 push.

Даже если в реальности он будет происходит несколько медленнее, чем на рис. 5.19, предварительная загрузка все же может быть рассмотрена в качестве безопасного промежуточного звена до тех пор, пока разработчики не найдут более эффективного способа безопасного использования HTTP/2 push, при котором не возникнет риск потери ресурсов.

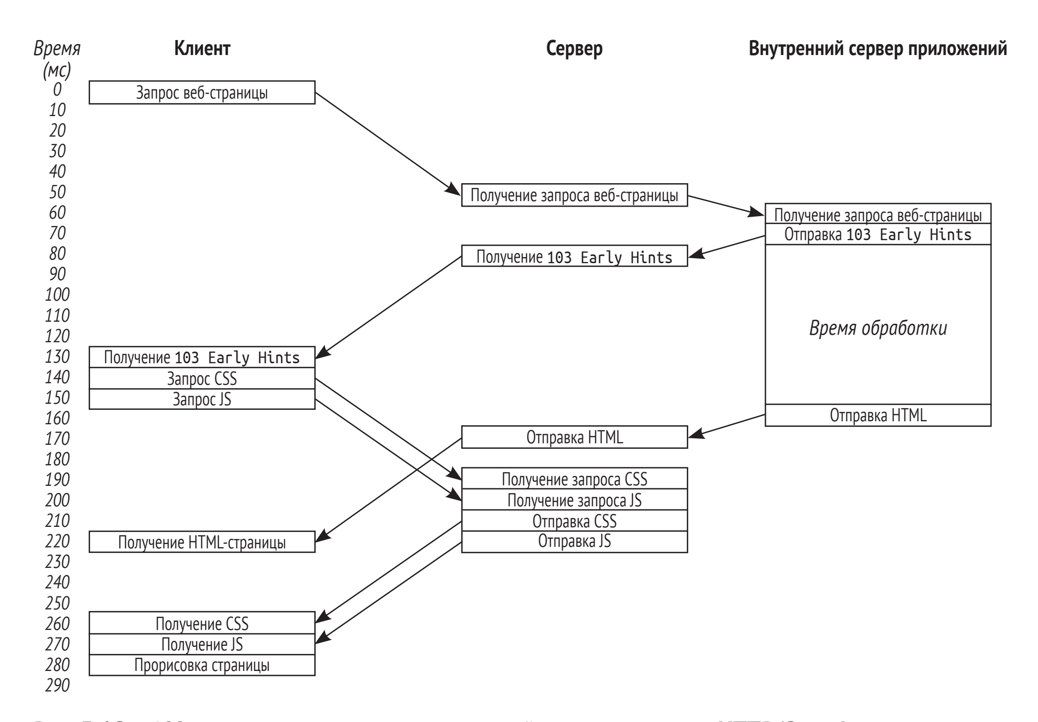


Рис. 5.19 **103 с заголовками предварительной загрузки вместо HTTP/2 push**

На момент написания этой книги ни один браузер не поддерживает обработку `103 Early Hints` (хотя Chrome[1] и Firefox[2] работают над этим). Также не все браузеры поддерживают заголовки ссылок предварительной загрузки[3]. Возможно, к тому времени, когда заголовок `103 Early Hints` будет поддерживаться повсеместно, проблема излишней push-загрузки решится, и, возможно, с помощью дайджестов или подобных методов; в таком случае у разработчиков будет два варианта.

5.9 Другие варианты использования HTTP/2 push

Сегодня у HTTP/2 push есть конкретное предназначение: заблаговременная отправка критически важных ресурсов для ускорения загрузки страницы без необходимости встраивания ресурсов. Однако многие

---

1 https://crbug.com/671310.
2 https://bugzilla.mozilla.org/show_bug.cgi?id=1407355.
3 https://caniuse.com/#feat=link-rel-preload.

люди обсуждают и другие варианты его использования[1]. Среди них рассматриваются следующие:

- *может ли HTTP/2 push заменить WebSockets или SSE, если требование использовать push только в ответ на определенные запросы было смягчено?* WebSockets и SSE обеспечивают двустороннюю связь между клиентом и сервером (например, она важна при обновлении веб-страницы или появлении новой информации на сервере). Или, может быть, достаточно использовать эти две технологии совместно с HTTP/2[2]? Как уже было сказано в начале главы, HTTP/2 push не может стать полноценной заменой, однако при условии внесения некоторых изменений, которые сделают связь между клиентом и сервером действительно двусторонней, у него есть потенциал (однако это может быть более накладно, по сравнению с исходным форматом, скажем, WebSockets). В связи с этим отдел исследований и разработок BBC рассмотрел использование HTTP/2 push в качестве метода широковещательной рассылки и опубликовал довольно интересную статью[3];
- *можно ли использовать HTTP/2 push для обновления кешей браузера при изменении ресурсов?* На сегодняшний день кеширование и очистка кеша – это довольно сложные процессы[4]. Но если бы HTTP/2 push позволял вам помещать ресурсы непосредственно в HTTP-кеш (что невозможно сейчас), появились бы новые возможности обработки изменений на веб-сайтах;
- *можно ли применить HTTP/2 push для концепции прогрессивных JPEG-файлов? Прогрессивные JPEG-файлы* загружаются браузером постепенно, т. е. по мере загрузки большего количества файлов эти изображения становятся четче. Такая концепция позволяет загружать несколько изображений параллельно. Если бы с помощью HTTP/2 push мы могли бы изменить приоритет этих файлов, было бы еще интереснее[5]. Таким образом, сервер мог бы отправлять предпросмотр изображения с высоким приоритетом, а затем уже с низким приоритетом отправлять основное изображение. В главе 7 мы обсудим веб-сервер Shimmercat, который как раз-таки использует эту технологию;
- *можно ли использовать HTTP/2 push для API?* Один из разработчиков API предположил, что HTTP/2 push можно использовать для отправки дополнительной информации и при этом сохранять разделение ресурсов, вследствие чего может возникнуть множество интересных вариантов использования, которые, возможно, не будут ограничены push-кешами браузеров. Протокол позволяет использовать push

---

1 https://www.igvita.com/2013/06/12/innovating-with-http-2.0-server-push/.

2 https://www.infoq.com/articles/websocket-and-http2-coexist.

3 https://www.bbc.co.uk/rd/publications/whitepaper336.

4 https://css-tricks.com/strategies-for-cache-busting-css/.

5 https://calendar.perfplanet.com/2016/even-faster-images-using-http2-and-progressive-jpegs/.

в ответ на любой запрос. Однако во многих отношениях браузеры ограничивают использование push и уведомляют об этом страницу, если впоследствии она не запрашивает ресурсы. Клиент HTTP/2, основанный не на браузере, может снять такие ограничения;

- *может ли добавление уведомлений способствовать появлению других вариантов использования HTTP/2 push?* Добавление push-уведомлений HTTP/2 или API в браузеры может привести к другим интересным вариантам использования. Например, новостной веб-сайт или веб-сайт социальной сети может проверить сервер на наличие обновлений с помощью одного короткого запроса «Есть ли обновления?». Если они есть (например, на сайт выложили новости), любые необходимые ресурсы могут быть переданы как стандартные ресурсы HTTP (HTML, данные JSON, изображения и т. д.); затем в веб-приложение должно быть отправлено событие, сообщающее клиенту о необходимости извлечения и последующего отображения этих ресурсов по их получении. Такая техника позволит загружать веб-страницы мгновенно. Преимущества этого метода перед WebSockets или SSE – это преимущества HTTP (например, кеширование, форматы файлов и простота).

Таким образом, у HTTP/2 push есть и другие полезные функции. Я думаю, что, несмотря на то что он не использовался по своему первоначальному предназначению (по веским причинам, обсуждаемым в этой главе), он имеет большой потенциал, который станет основой для развития новых идей (упомянутых в этой главе и, возможно, каких-либо новых[1]). Инженерная группа Internet (IETF) начала разработку информационного RFC, который будет включать в себя варианты использования HTTP/2 push[2].

Не слишком ли мы ограничиваем себя, пытаясь закрепить за HTTP старую концепцию протокола запроса–ответа? WebSockets и SSE демонстрируют потребность в двустороннем протоколе на основе HTTP, и, возможно, нам следует это реализовать. На этот счет уже поступали предложения[3], и уровень двоичных фреймов, представленный в HTTP/2, позволяет реализации такого рода. В главе 10 я еще вернусь к обсуждению этого тезиса.

Так или иначе, HTTP/2 push является новинкой, и владельцы сайтов должны подходить к ней с осторожностью. У этой технологии есть большой потенциал, и в дальнейшем эксперименты покажут, действительно ли HTTP/2 push обеспечивает прирост производительности, или же, наоборот, слишком усложняет процесс и в принципе не стоит усилий.

Я надеюсь, что после прочтения этой главы вы увидели, что, несмотря на то что потенциал HTTP/2 push действительно огромен, торопиться с его реализацией не стоит. На самом деле вы можете не рассматривать

---

1 https://groups.google.com/a/chromium.org/forum/#!msg/net-dev/yfkW4mkWIPU/5RckmfktJgAJ; https://goo.gl/gTJrwC.

2 https://tools.ietf.org/html/draft-bishop-httpbis-push-cases.

3 https://tools.ietf.org/html/draft-benfield-http2-p2p.

его реализацию до тех пор, пока не будет предложено больше передовых практик и методов с учетом кеширования.

Резюме

- HTTP/2 push – это новая концепция HTTP/2, которая позволяет отправлять несколько ответов на один HTTP-запрос.
- HTTP/2 push может стать альтернативой концепции встраивания критических ресурсов.
- Многие серверы и сети CDN реализуют HTTP/2 push посредством заголовков HTTP-ссылок.
- Новый код состояния `103 Early Hints` может помочь заблаговременно предоставлять заголовки ссылок.
- HTTP/2 push может быть реализован в браузере довольно неочевидными способами.
- Зачастую возникающая избыточная push-загрузка ресурсов может отрицательно сказаться на производительности веб-сайта.
- Прирост производительности от HTTP/2 push может быть не велик, а риски высоки.
- Возможно, лучше пользоваться технологией предварительной загрузки (с `103 Early Hints`, если вы хотите пользоваться push).
- HTTP/2 push можно использовать для других целей, хотя некоторые из них потребуют изменений в протоколе.

6 Оптимизация в HTTP/2

В этой главе мы рассмотрим:

- значение HTTP/2 для веб-разработчиков;
- методы повышения веб-производительности HTTP/1.1 и являются ли они антипаттернами для HTTP/2;
- другие методы повышения производительности и их актуальность для HTTP/2;
- оптимизацию в HTTP/1 и HTTP/2;
- метод слияния соединений.

Итак, вы имеете четкое представление о HTTP/2: вы знаете, для чего он предназначен и как работает, а также знаете новые функции и новые возможности, которые открываются благодаря нему. В третьей части этой книги я расскажу о более сложных аспектах протокола, однако сейчас ваших знаний уже достаточно, для того чтобы понять, что HTTP/2 значит для ваших веб-сайтов и как вы можете их оптимизировать. Как следует изменить методы разработки? Возможно ли отказаться от некоторых техник исполнения? Какие новые техники вы можете использовать? Как оптимизировать работу сайта для тех пользователей, которые не поддерживают HTTP/2? Данная глава призвана ответить на все эти вопросы.

6.1 Значение HTTP/2 для веб-разработчиков

Я думаю, вы заметили, что HTTP/2 в корне меняет способ отправки HTTP-сообщений на сервер, ввиду чего и повышается производительность. Но нужно ли разработчикам менять свои языки и методы разработки? Следует ли использовать определенные фреймворки JavaScript с целью получить все преимущества HTTP/2? Ответить на эти вопросы можно следующим образом: в целом никаких изменений не требуется, хотя некоторые из них все же могут быть полезны.

Протокол HTTP/2 имеет обратную совместимость. Если ваш сервер поддерживает HTTP/2, у вас будет возможность включить его и в большинстве случаев вы сразу увидите прирост производительности, и при этом ни одна строчка кода не поменяется. Изменений требуют некоторые новые функции (такие как HTTP/2 push). Вы сможете реализовать эти изменения после тщательного изучения HTTP/2, в чем вам может помочь в том числе и эта глава. Однако HTTP/2 не будет требовать внесения подобных изменений сразу же после перехода на него. По большей части изменения являются необязательными дополнениями, которые могут обеспечить еще больший прирост производительности.

Конечно же, переход на HTTP/2 может оказаться не таким простым, как хотелось бы (об этом мы говорим в главе 3 и в приложении). Возможно, при переходе вам потребуется обновить свою инфраструктуру или даже добавить новые ее элементы, например разместить перед вашим веб-сервером обратный прокси-сервер или сеть доставки контента. Такие новшества могут повлечь за собой новые возможности, которыми вы сможете воспользоваться в будущем или при ближайшем обновлении. Однако данная тема требует отдельного рассмотрения. В этой главе я попытаюсь объяснить, каково значение HTTP/2 для веб-разработчиков (если предположить, что они имеют доступ к нему).

HTTP/2-запросы от браузера к серверу

Одно из главных преимуществ HTTP/2 заключается в том, что, если сервер уже перешел на поддержку этой версии, на стороне клиента не требуется внесения никаких изменений; браузер сам позаботится об этом. Вам не нужно обновлять версию JQuery, использовать другой синтаксис AJAX или переключаться с Angular на React (или наоборот). С точки зрения веб-разработчика, HTTP-запросы и ответы с клиентской стороны работают точно так же, как и раньше. Единственным изменением здесь в идеале должна являться скорость: ввиду отсутствия очередей, все происходит намного быстрее. Библиотеки и инструменты позволяют браузеру обрабатывать детали совершения запроса на низком уровне. Таким образом, знать о HTTP/2 достаточно только браузеру.

На данный момент разработчики пользовательских интерфейсов не могут дать однозначный ответ на вопрос, следует ли использовать HTTP/1.1 или HTTP/2, как и не могут точно сказать, следует ли использовать HTTP/1.1

или HTTP/1.0 (или даже HTTP/0.9). В будущем ситуация может измениться, поскольку, например, возможность расстановки приоритетов запросов в HTML или AJAX[a,b,c] или реализация обратных вызовов HTTP/2 push[d] – чрезвычайно полезные функции.

[a] https://bugzilla.mozilla.org/show_bug.cgi?id=559092.
[b] https://bugs.chromium.org/p/chromium/issues/detail?id=41501.
[c] https://github.com/WICG/priority-hints.
[d] https://github.com/whatwg/fetch/issues/65.

6.2 Оптимизация HTTP/1.1 мешает HTTP/2?

Целью HTTP/2 является решение фундаментальных проблем производительности HTTP/1.1. Именно из-за этих проблем запрос отдельных ресурсов в HTTP/1.1 был очень затратным, ввиду чего и появились многочисленные методы, связанные с увеличением количества HTTP-соединений или минимизацией количества запрашиваемых ресурсов. В первом случае браузеры открывали несколько соединений, а иногда даже размещали ресурсы в нескольких доменах (сегментирование). Во втором случае для объединения файлов CSS и JavaScript применялась технология конкатенации, вследствие чего они объединялись в один большой файл. Изображения небольших размеров (такие как иконки социальных сетей или другие маленькие значки) также объединялись в спрайт-файлы, из которых элементы можно было извлекать посредством CSS. Оба типа оптимизации предполагали передачу одних и тех же данных (или, по крайней мере, похожих данных) посредством выполнения меньшего количества HTTP-запросов.

Несмотря на то что вышеописанные обходные пути устраняют некоторые недостатки HTTP/1.1, в то же время они создают и проблемы (описанные в главе 2). HTTP/2 призван решить многие из них на уровне протокола. В новой версии выполнение запросов почти не требует затрат ввиду внедрения технологии двоичного фрейминга. Встает вопрос, актуальны ли в таком случае обходные пути? На самом деле, многие говорили, что для HTTP/2 они становятся помехами. Но не все так просто; я сказал, что запросы HTTP/2 *почти* не требуют затрат и *только* на уровне протокола.

6.2.1 Запросы HTTP/2 по-прежнему затратны

Когда веб-страница ссылается на ресурс, происходит множество разных процессов; некоторые из них оптимизированы в HTTP/2, а некоторые – нет. На рис. 6.1 подробно описаны некоторые из множества решений и процессов, которые браузер должен осуществить, когда веб-страница запрашивает ресурс.

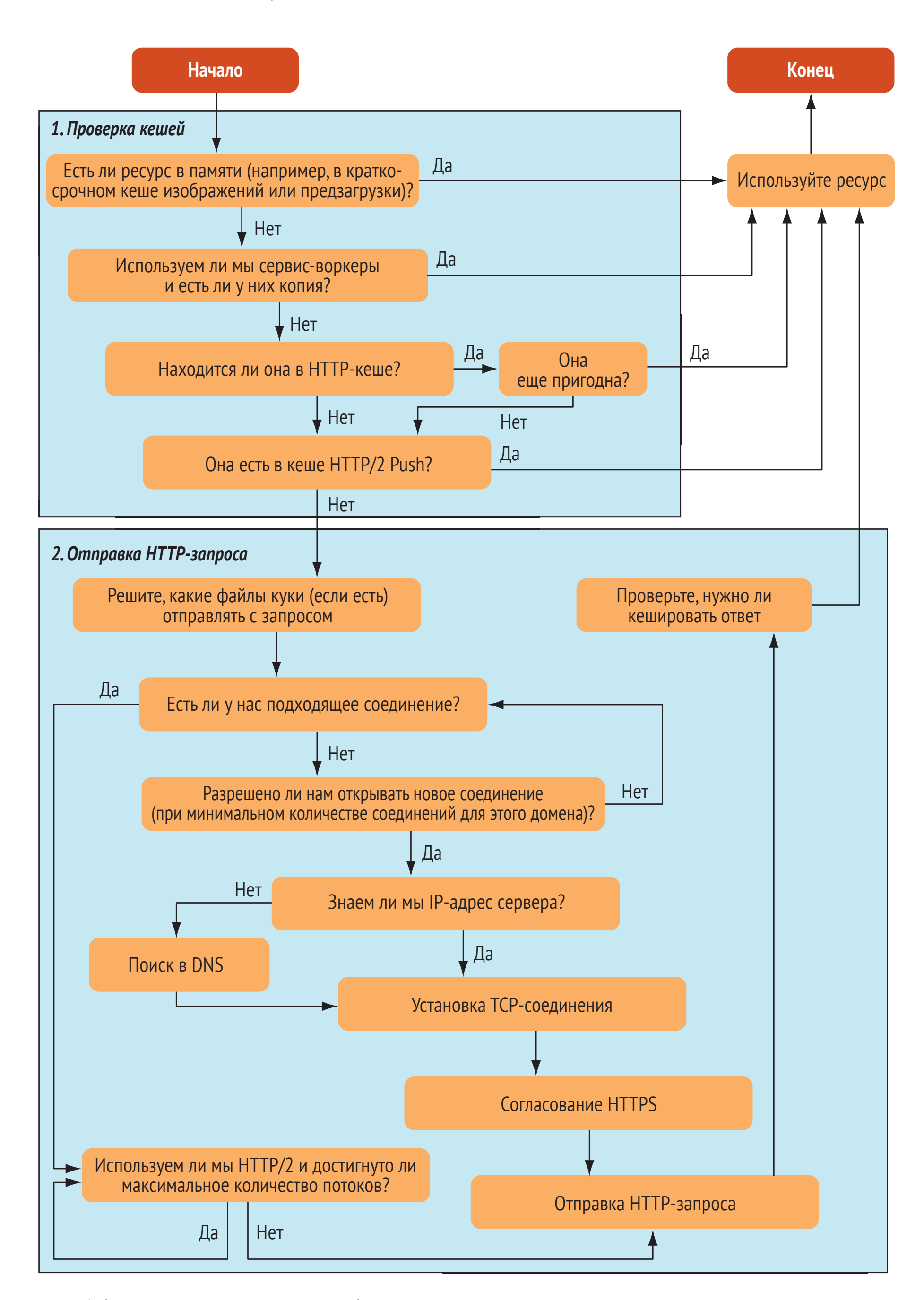


Рис. 6.1 Решения и процессы браузера, связанные с HTTP-ресурсом

На высоком уровне браузеру сначала необходимо проверить наличие актуальной копии в кешах (которые мы обсуждали в главе 5), и если она есть, то он уже не будет выполнять для этого ресурса новый запрос. В за-

висимости от используемого домена и типа запроса, HTTP-запрос может выполняться по уже существующему соединению или же по-новому. После загрузки ресурса клиенту необходимо просмотреть заголовки кеширования, чтобы решить, сохранять ли ресурс в кеш для потенциального повторного использования. Даже после всех этих шагов браузер должен обработать ресурс (проанализировать CSS или JavaScript, обработать изображение JPEG и т. д.).

Множество процессов происходит еще до того, как вы доходите до стадии отправки и получения HTTP-запросов, что уж говорить о том, когда дело доходит до получения ресурса. Все эти процессы требуют времени, но зачастую совсем немного. Однако, если вы отойдете от ограничения HTTP/1.1, разрешающее только шесть параллельных подключений, и получите сотни ресурсов, вы заметите, что возникли новые интересные проблемы.

В 2016 году, когда использование HTTP/2 набирало обороты, разработчики Chrome заметили значительные задержки при использовании HTTP/2 для большого количества ресурсов[1]. Например, даже при отсутствии запросов некоторые сайты зависали на 400–500 мс. Однако эта проблема не имела ничего общего с протоколом HTTP/2 как таковым. В главе 7 вы увидите, что приоритизация потоков должна позволять параллельным запросам распределять соответствующие приоритеты на уровне протокола. Проблема заключалась исключительно в том, что «узким местом» системы была именно отправка ресурсов. В это время браузер был занят выполнением всех задач, упомянутых в предыдущем разделе. Благодаря HTTP/2 множество ресурсов теперь могут работать одновременно, и при этом проблем с производительностью, связанных с ожиданием свободного соединения, не возникает; однако эта возможность перемещает «узкое место» в другие части системы.

Чтобы исключить эту задержку, команде Chrome пришлось ограничить количество ресурсов, которые могут быть поставлены в очередь даже при использовании HTTP/2, вернувшись к шести подключениям, как в HTTP/1. Так продолжалось до тех пор, пока команда не оптимизировала код. Ограничение применялось только к несущественным ресурсам, поэтому в теории HTTP/2 ничем не уступал HTTP/1.1. Однако возник интересный сценарий: ограничение распространялось на ресурсы JavaScript с атрибутами `async` или `defer` (обычно добавляемыми из соображений производительности, так как они предотвращают блокировку ресурсов), а ресурсы JavaScript без этих атрибутов не ограничивались. Веб-сайты, которые использовали передовые методы повышения производительности (например, неблокируемые JavaScript), были искусственно ограничены и загружались медленнее, чем сайты, не использующие их. В конечном счете веб-сайты, использующие `async` или `defer`, загружались точно так же, как запросы HTTP/1.1 (см. рис. 6.2).

Вопрос о значении этой проблемы остается открытым; если ресурс JavaScript был помечен атрибутами `async` или `defer`, значит, он не являет-

[1] https://bugs.chromium.org/p/chromium/issues/detail?id=655585.

ся критическим для загрузки. Однако для владельцев веб-сайтов, реализовавших HTTP/2.2, такие исходы, свойственные HTTP/1, стали большим сюрпризом[1].

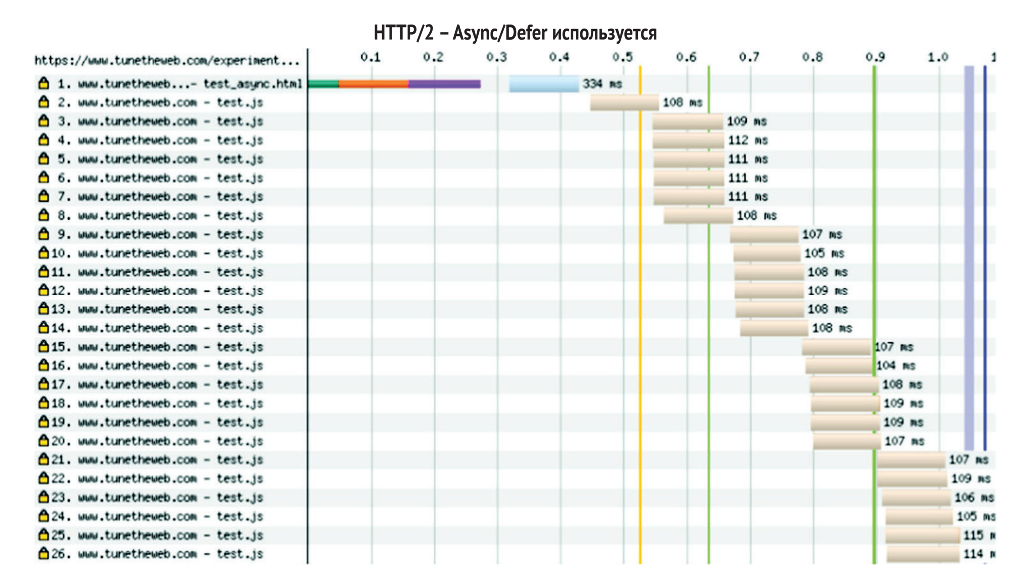


Рис. 6.2 Ресурс JavaScript, не блокирующий рендеринг, искусственно замедляется в Chrome

В более поздних версиях Chrome проблема ограничения была решена. Она стала ярким примером того, что при устранении «узких мест» раз-

[1] https://stackoverflow.com/questions/45384243/google-chrome-does-not-do-multiplexing-withhttp2/45775288#45775288.

работчикам необходимо быть осторожными, поскольку за ними могут скрываться и другие проблемы с производительностью.

6.2.2 Возможности HTTP/2 не безграничны

Следует также отметить, что в HTTP/2 ограничения касательно количества соединений сняли не полностью. В главе 5 мы увидели, что, несмотря на то что по умолчанию `SETTINGS_MAX_CONCURRENT_STREAMS` имеет значение unlimited, во многих реализациях все же существуют ограничения (см. табл. 6.1 и 6.2).

Таблица 6.1 Ограничения количества параллельных потоков в популярных серверных реализациях HTTP/2

Программное обеспечение	Тип	Допустимое по умолчанию количество параллельных потоков
Apache HTTPD (v2.4.35)	Веб-сервер	100[a]
nginx (v1.14.0)	Веб-сервер	128[b]
H2O (2.3.0)	Веб-сервер	100[c]
IIS (v10)	Веб-сервер	100
Jetty (9.4.12)	Веб-контейнеры сервлетов Java	128[d]
Apache Tomcat (9.0)	Веб-контейнеры сервлетов Java	200[e]
Node (10.11.0)	Операционная среда JavaScript	100[f]
Akamai	Сеть CDN	100
Amazon CloudFront и S3	Сеть CDN	128
Cloudflare	Сеть CDN	128
MaxCDN	Сеть CDN	128

[a] https://httpd.apache.org/docs/2.4/mod/mod_http2.html#h2maxsessionstreams.
[b] http://nginx.org/en/docs/http/ngx_http_v2_module.html#http2_max_concurrent_streams.
[c] https://h2o.examp1e.net/configure/http2_directives.html#http2-max-concurrent-requests-per-connection.
[d] https://www.eclipse.org/jetty/documentation/9.4.x/http2-configuring.html.
[e] https://tomcat.apache.org/tomcat-9.0-doc/config/http2.html.
[f] https://nodejs.org/api/http2.html.

Таблица 6.2 Ограничения количества параллельных потоков в популярных браузерах с поддержкой HTTP/2

Программное обеспечение	Допустимое по умолчанию количество параллельных потоков
Chrome (v69)	1000
Firefox (v62)	Не задано (HTTP/2 используется по умолчанию без ограничений)
Safari (v12)	1000
Opera (v56	1000
Edge (v17)	1024
Internet Explorer 11	1024

Такие значения были приняты по одной из двух причин: либо они рекомендованы в документации, либо их выявили в ходе экспериментов. Сети CDN я исследовал с помощью Wireshark, посредством которой пере-

хватывал запросы на домашнюю или другую страницу этой CDN, и просматривал *фрейм* SETTINGS, как описано в главе 4 (раздел 4.3.3). С браузерами я работал с помощью сервера nghttp, настроив его на подробное ведение журнала; далее я также просматривал фреймы SETTINGS, отправляемые браузерами при установке соединения HTTP/2.

Сразу бросается в глаза один факт: значения на стороне сервера намного меньше значений для веб-браузеров. В Firefox ограничение в принципе не установлено, и по умолчанию здесь может быть открыто неограниченное количество параллельных соединений. Это имеет смысл, поскольку в конечном итоге браузер получает полный контроль над отправляемыми запросами, ввиду чего он может добавлять ограничения вне протокола (как изначально было у Chrome, который мы обсуждали в предыдущем разделе). Кроме того, количество запросов, обслуживаемых браузером, должно быть гораздо меньше, в отличие от сервера, который, как правило, одновременно взаимодействует сразу с множеством пользователями. Несмотря на все это, учитывая проблемы с производительностью, с которыми сталкивался Chrome при параллельном запросе большого количества ресурсов, все же разумнее было бы ввести ограничения, хотя бы до тех пор, пока HTTP/2 не будет полностью адаптирован. Сайты, состоящие из более 200 ресурсов, встречаются редко. Однако после полной адаптации HTTP/2 и при условии прекращения объединения веб-сайтами ресурсов в спрайт-файлы количество таких сайтов непременно возрастет. Например, если веб-сайту требуется 500 ресурсов и они не объединены в спрайт-файлы, вполне вероятно, что сначала начнут загружаться 100–128 ресурсов, а остальные будут поставлены в очередь, как это было в HTTP/1.1. Мы уже встречались с подобной ситуацией в главе 2 (раздел 2.6.1), когда обсуждали превышение предела сервера в 100 потоков. Существует множество примеров, когда сайты отказывались от объединения файлов, выходя за пределы этих ограничений[1].

6.2.3 Для больших ресурсов эффективнее сжатие

Все веб-ресурсы перед отправкой по сети должны быть сжаты. У некоторых типов ресурсов (например, JPEG- и PNG-изображения и WOFF, и WOFF2 для шрифтов) сжатие встроено в формат. Для текстовых форматов, таких как HTML, CSS и JavaScript, используется сжатие, такое как gzip (или более новый вариант brotli[2]), которое выполняется «на лету».

Общей чертой вышеперечисленных алгоритмов является то, что файлы большего размера они сжимают более эффективно, чем файлы меньшего размера. В разделе 7.3.4 описана работа каждого алгоритма. Если сказать кратко, большинство из них работает по следующему принципу: они ищут повторяющиеся последовательности данных и заменяют их ссылками на версию этих данных. В больших файлах обычно встречает-

---

1 http://engineering.khanacademy.org/posts/js-packaging-http2.htm#http-2-0-has-service-issues.

2 https://tools.ietf.org/html/rfc7932.

ся больше дубликатов, а степень сжатия обычно больше. Однако лучше сжимать один большой файл размером 100 Кб, чем несколько файлов по 10 Кб по отдельности, даже если в общей сумме объемы несжатых данных одинаковы. Отказавшись от объединения ресурсов (как в случае с HTTP/1.1), вы можете потерять некоторые преимущества сжатия, и в конечном итоге по протоколу HTTP/2 вы будете отправлять по сети *больше* данных даже при одинаковом объеме файлов.

Также ситуация зависит и от типа сжимаемых файлов. Рассмотрим на примере загрузки популярных файлов jQuery:

```
curl -OL https://code.jquery.com/jquery-3.3.1.min.js
curl -OL https://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.js
curl -OL https://code.jquery.com/ui/1.12.1/jquery-ui.min.js
```

Затем объединим эти файлы с помощью команды Linux `cat`:

```
cat jquery-3.3.1.min.js jquery.mobile-1.4.5.min.js jquery-ui.min.js >
jquery_combined.js
```

Взглянув на список файлов, мы можем увидеть их размеры:

```
-rw-r--r-- 1 barry p  86927 19 May 19:31 jquery-3.3.1.min.js
-rw-r--r-- 1 barry p 253668 19 May 19:31 jquery-ui.min.js
-rw-r--r-- 1 barry p 200143 19 May 19:31 jquery.mobile-1.4.5.min.js
-rw-r--r-- 1 barry p 540738 19 May 19:31 jquery_combined.js
```

Общий объем файла составляет 86 927 + 200 143 + 253 668 = 540 738 Кб. Затем мы сжимаем все файлы с помощью gzip со стандартными настройками:

```
gzip jquery*
```

Таким образом, мы получаем файлы следующих размеров:

```
-rw-r--r-- 1 barry p  30371 19 May 19:31 jquery-3.3.1.min.js.gz
-rw-r--r-- 1 barry p  68058 19 May 19:31 jquery-ui.min.js.gz
-rw-r--r-- 1 barry p  55649 19 May 19:31 jquery.mobile-1.4.5.min.js.gz
-rw-r--r-- 1 barry p 152652 19 May 19:31 jquery_combined.js.gz
```

Сложив значения (30 371 Кб + 68 058 Кб + 55 649 Кб), мы видим, что файл сжимается до 152 652 Кб. Объем становится меньше всего на 1 %, но разница все же есть, и для неоптимизированных файлов (которые я обсуждаю в разделе 6.3.1) она может быть значительнее.

С другой стороны, отказ от объединения файлов может позволить разработчикам более направленно выбирать содержимое страницы. Например, раньше некоторые разработчики могли использовать один большой объединенный файл JavaScript для каждой страницы сайта. Некоторым страницам требовались одни части этого файла, а некоторым другие; поскольку работа в HTTP/1.1 с несколькими файлами неэффективна, отправлять один большой файл для каждой страницы было удобнее, чем создавать отдельные пакеты. С появлением HTTP/2 одновременная загрузка множества файлов стала менее проблематичной. Сейчас для каждой страницы можно создать отдельный файл только с тем JavaScript,

который ей потребуется; таким образом, объем данных, передаваемых для каждой страницы уменьшается.

Компенсирует ли сокращение передаваемых данных потерю степени сжатия? Это зависит от вашего сайта и от того, разделяете ли вы файлы и насколько меньше ресурсов вы стали отправлять. В ходе тематических исследований[1] наблюдалось небольшое увеличение объема данных, передаваемых по сети даже при удалении ненужного кода, хотя огромное количество передаваемого кода JavaScript вызывает более заметные проблемы. Даже если объем ресурсов несколько больше, прирост производительности ввиду обработки меньшего количества данных на некоторых страницах может компенсировать потерю производительности из-за отправки большего количества необработанных байтов при использовании разделения. Также HTTP/2 уже не требует такого большого количества пакетов, как это было HTTP/1.1, но в любом случае вам стоит учитывать потери степени сжатия перед разделением.

6.2.4 Ограничение пропускной способности и конкуренция ресурсов

HTTP/1.1 передает запросы независимо от их приоритета, вследствие чего возникают сложности, ведь самые разные запросы могут быть инициированы в самое разное время. Именно поэтому разработчики веб-браузеров начали активную работу над концепцией приоритетности запросов, с помощью которой наиболее важные из них отправлялись бы в первую очередь. Обходными путями для этой проблемы были объединение и сегментирование ресурсов, однако эти решения имели множество ограничений. Сегодня, когда ограничения минимальны (или даже вовсе устранены), может возникать конкуренция ресурсов за пропускную способность соединения.

На сайте-примере из главы 2 при загрузке 360 изображений по HTTP/2 100 из них загружались параллельно. Здесь мы видим явные отличия от HTTP/1.1 и его ограничение в шесть параллельно загружаемых ресурсов. Поскольку 100 загрузок происходили одновременно, загрузка каждого ресурса по протоколу HTTP/2 занимала больше времени, как показано на рис. 6.3.

Вообще, по HTTP/2 изображения загружаются намного быстрее. Однако было бы лучше, если вместо постепенной загрузки каждого изображения (как в HTTP/2, если не используется приоритизация) сначала осуществлялась бы полная загрузка некоторых ресурсов (как это происходит в HTTP/1.1).

С вышеописанными проблемами при переходе на HTTP/2 столкнулся веб-сайт, посвященный дизайну, 99design[2]. После перехода у него появи-

1 http://engineering.khanacademy.org/posts/js-packaging-http2.htm#bundling-improves-compression.

2 https://99designs.com/tech-blog/blog/2016/07/14/real-world-http-2-400gb-of-images-per-day/.

лась возможность загружать все необходимые веб-странице ресурсы параллельно, в том числе и большие изображения высокого качества (что очень важно для веб-сайта о дизайне). Однако некоторые изображения из верхней части страницы загружались медленнее, поскольку в это же время загружались и изображения из нижней части страницы, занимая при этом пропускную способность. В HTTP/1.1 браузер отдавал приоритет изображениям, которые должны отобразиться на экране первыми, а ограничение в шесть подключений гарантировало, что их загрузка не будет замедляться загрузкой множества последующих изображений с более низким приоритетом.



Рис. 6.3 Загрузка отдельных ресурсов по HTTP/1.1 (сверху) происходит быстрее, чем по HTTP/2 (снизу)

Данные проблемы решаются правильной установкой приоритетов запросов. До этого я рассказывал о приоритизации лишь кратко, но в главе 7 я рассмотрю этот аспект подробно. Приоритизация не зависит от владельца сайта (поэтому я оставил ее детальное обсуждение для следующей главы), но она может оказаться ключевым фактором, влияющим на производительность как браузеров, так и серверов. Данный способ наиболее эффективен, если браузер может предложить соответствующий приоритет (например, загрузка изображений в области просмотра с приоритетом выше, чем у изображений, расположенных дальше по странице) и если сервер может ответить на это предложение. В качестве альтернативы изображения могут загружаться полностью, но строго в порядке следования запросов. Так, например, делает Chrome (см. главу 7).

Следовательно, эта проблема была скорее проблемой ранней реализации, чем проблемой самого протокола HTTP/2. Однако, учитывая, что разработчики имеют 20-летний опыт борьбы с проблемами в HTTP/1, маловероятно, что это станет последней подобной проблемой при переходе к HTTP/2.

6.2.5 Сегментирование данных

Сегментирование данных использовалось для преодоления накладываемого браузерами на домен ограничения в шесть параллельных соединений. Количество параллельных загрузок увеличивали за счет размещения медиафайлов на поддоменах или в отдельных доменах. На мой взгляд, полезный эффект от сегментирования был преувеличен, за исключением ситуаций, где необходимо загрузить большое количество ресурсов, и в этом случае целесообразно было бы применять *объединение* или *спрайтинг*. Исследования показали, что зачастую дополнительные соединения создаются только для одного или двух ресурсов, поэтому время, затраченное на установку этих соединений, может свести на нет любой возможный прирост производительности при их эффективном использовании. И как всегда, разработчикам необходимо оценить степень влияния таких технологий, как сегментирование, на их сайт, а не слепо применять их, предполагая, что они обязательно обеспечат прирост производительности.

С появлением HTTP/2 сегментирование утратило смысл, а попытки создания отдельной инфраструктуры и управления ей теперь приносят ограниченные результаты. Кроме того, некоторые технологии HTTP/2 (например, *HTTP/2 push* и сжатие заголовков HPACK) лучше работают при наличии только одного соединения, поэтому сегментирование приведет здесь к снижению производительности. Я считаю, что, в силу того что популярность HTTP/2 набирает стремительные обороты, с течением времени сегментирование будет использоваться все реже. В разделе 6.4.4 вы увидите, что для HTTP/2 существует метод обратного сегментирования, объединяющий лучшие черты двух версий протокола. При некоторых конкретных сценариях, где потери очень высоки, сегментирование все еще может быть полезно (см. главу 9). Однако по большей части

сегментирование уже утратило свою ценность. Даже в случае подобных сценариев, браузерам лучше открывать несколько соединений только в случае необходимости, а не оставлять установку сегментирования для всех подключений (независимо от того, с потерями они или нет) на усмотрение сайтов.

6.2.6 Встраивание ресурсов

Я всегда считал встраивание критически важного CSS или скриптов чем-то вроде хакерства[1] – эффективного, но тем не менее хакерства. Встраивание кода CSS в заголовок страниц ускоряет загрузку первой страницы, но затем этот код либо дублирует содержимое реального CSS, либо блокирует кеширование для последующих загрузок страниц на том же сайте. Кроме того, встраивание исключительно критического CSS может быть сложным осуществить, так как по умолчанию уже может быть предусмотрен определенный способ загрузки CSS с целью предотвращения блокировки их рендеринга.

Предполагалось, что *HTTP/2 push* устранит необходимость встраивания ресурсов, но, как мы увидели в главе 5, эффективное использование данной технологии оказалось довольно сложным, и поэтому она не получила должного распространения. Я предполагаю, что еще какое-то время встраивание будет оставаться неплохим способом повышения производительности для тех веб-сайтов, которые готовы его использовать и хотят выжать из загрузки первой страницы все до последней капли.

6.2.7 Заключение

Одной из основных целей HTTP/2 было решение проблемы повышенной ресурсоемкости запросов в HTTP/1.1. HTTP/2 ввел значительные улучшения, однако HTTP-запросы по-прежнему требуют затрат. Часто эта проблема возникает из-за проблем вне протокола (таких как высокая ресурсоемкость при множественных запросах к браузерам), но другие, по крайней мере частично, связаны именно с HTTP/2. Например, при поступлении слишком большого количества одновременных запросов может произойти замедление некоторых показателей (таких как первичная отрисовка), если для критических запросов ресурсов не хватает.

Также необходимо учитывать факт того, что многие реализации HTTP/2 еще не созрели; но, несомненно, некоторые недоработанные реализации со временем будут оптимизированы. Кроме того, могут быть обнаружены ошибки или протокол может использоваться неэффективно (например, при неправильной приоритизации запросов). Я настоятельно рекомендую обновить программное обеспечение HTTP/2 (серверам и браузерам) именно сейчас.

Повторяя предыдущее высказывание, я могу сказать, что по прошествии 20 лет разработчики отлично разбираются в HTTP/1 и зрелых

[1] https://www.tunetheweb.com/blog/inlining-css-is-not-for-me/.

технологических стеках, но HTTP/2 все еще находится в зачаточном состоянии. Может показаться, что у новой версии вероятность возникновения проблем, затрудняющих показ страницы, не так высока, однако существует множество «узких мест» и других примеров, где HTTP/2 не так эффективен, как мог бы быть.

В HTTP/2 резко увеличились лимиты параллельной загрузки (по сравнению с былым ограничением в шесть соединений), однако отойти от них полностью все еще не удается. Пока что я рекомендую параллельно загружать не более 100 ресурсов на один домен. При этом не отказывайтесь от концепции объединения ресурсов, даже если она приведет к созданию сотен файлов; установите повышенные ограничения для создания соединения должным образом.

Потребность в оптимизации производительности, идея которой преобладала в HTTP/1.1, сейчас намного меньше; однако до тех пор, пока разработчики не привыкнут к работе с новой версией протокола, полностью отказываться от подобных мер не стоит. Вместо этого вы можете просто сократить их использование до подходящего вам уровня (например, просто объединять меньше ресурсов, а не отказываться от их объединения вовсе). Разработчикам лучше сгруппировать код в наборы файлов, которые с большей вероятностью будут использоваться совместно, а не объединять все в один большой файл, как они, возможно, делали в прошлом. Сокращение использования методов HTTP/1.1, а не их искоренение, является консенсусом, к которому рано или поздно приходят исследователи и владельцы веб-сайтов[1]. После появления HTTP/2 многие считали, что старые методы стали неуместны, но это не совсем так.

Ключевой вывод – не стоит предполагать, что HTTP/2 являет собой чудодейственное средство, которое может избавить вас от всех возникающих проблем. Новая версия может подойти для одних сайтов, но не подойти для других. Вам не обойтись без тщательного изучения и разбора протокола.

6.3 Методы повышения веб-производительности все еще актуальны для HTTP/2

В этой главе мы рассмотрели, что HTTP/2 действительно устраняет некоторые недостатки более ранних версий протокола, вследствие чего некоторые методы повышения производительности становятся для него менее актуальными. Оптимизация протокола HTTP – это не единственный путь повышения производительности веб-сайтов, но, в соответствии с природой Internet, которая предполагает взаимодействие клиента и сервера на расстоянии, большая часть работы веб-сайта должна быть оптимизирована именно на сетевом уровне. Стоит рассмотреть также

[1] https://uhdspace.uhasselt.be/dspace/bitstream/1942/23909/1/h2bestpractices_RobinMarx_WEBIST2017.pdf.

другие передовые техники передачи данных и объяснить, почему они все еще актуальны для HTTP/2, а также рассмотреть новые возможности в HTTP/2.

6.3.1 Уменьшение объема передаваемых данных

Независимо от того, насколько быстро HTTP/2 позволяет обрабатывать запросы и ответы, предпочтительнее отправлять меньшее количество данных. Он не способен чудесным образом ускорить соединение, он просто делает его более эффективным. Может показаться, что веб-сайты стали загружаться быстрее, однако количество загружаемых данных не изменилось (возможно, их стало даже немного больше, если сжатие не эффективно, как показано в разделе 6.2.3). Следовательно, объем данных необходимо сокращать настолько, насколько это возможно. Все прежние методы актуальны и для HTTP/2.

Использование подходящих форматов и размеров файлов

Пользователям всегда интереснее посещать веб-страницы с большим количеством мультимедиа, однако для загрузки и отображения медиафайлов требуется некоторое время. Согласно данным HTTP Archive, около 80 % размера страницы приходится на изображения и видео[1] (см. рис. 6.4), поэтому чрезвычайно важно учитывать целесообразность формата и размера отправляемых медиафайлов.

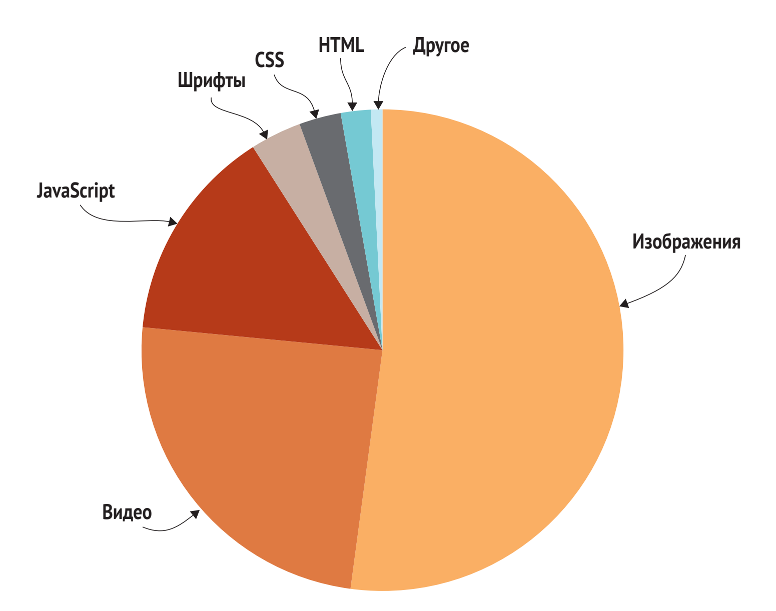


Рис. 6.4 Среднее количество байтов на страницу по типу контента

[1] https://httparchive.org/reports/page-weight.

Видео и аудио – это несколько специфический тип данных, поэтому рассматривать их здесь мы не будем, а вместо этого обсудим изображения[1]. Обычно фотографии имеют формат JPEG (он же JPG), а другие графические элементы – формат PNG. Google разработала особый формат сжатия изображений WebP, однако за несколько лет существования он так и не прижился[2]. Также часто используется формат SVG, однако над ним еще предстоит проделать работу. Почти все веб-сайты пользуются JPEG и PNG[3], поскольку эти форматы поддерживаются повсеместно и обеспечивают баланс между размером и качеством.

Формат JPEG несколько снижает качество изображений, однако он позволяет настроить баланс между качеством и уровнем сжатия. Вы можете снизить качество изображения и получить изображения гораздо меньшего размера, но без видимого падения качества для человеческого глаза. Изображения также могут содержать значительный объем метаданных (информация о том, когда была сделана фотография, на какую камеру она была сделана, какие настройки ISO использовались и т. д.). Большая часть этих метаданных не важна тем, кто просматривает вашу веб-страницу, и их следует удалять; но будьте осторожны с изображениями, которые вам не принадлежат, их изменения могут нарушить авторские права. Для изменения качества и метаданных изображения (что позволит вам уменьшить их размер) можно использовать различные инструменты и другое программное обеспечение для редактирования изображений, однако я советую пользоваться tinypng.com[4], так как он прост в использовании и сжимает JPEG и PNG быстро и без проблем. Также вы можете использовать tinyjpg.com. Данные сайты работают примерно одинаково и могут обрабатывать оба вышеуказанных формата. На рис. 6.5 показано, насколько уменьшается размер большинства изображений.

Следует учитывать не только уровень качества изображения, но и его размер. Например, бессмысленно отправлять изображения размером 5120×2880 пикселей для отображения его с шириной 100 пикселей, поскольку затрачивается слишком много времени для загрузки и обработки браузером. Никогда не размещайте на веб-страницах изображения большого размера с качеством или размером как для печати. Если необходимо сделать такие типы изображений доступными для пользователя, лучше оставьте на странице ссылки на их скачивание.

Для мобильных и полных версий вашего сайта (а также для разных размеров экрана) следует использовать изображения разных размеров. Конечно, посетители, у которых есть возможность просмотреть ваш сайт на большом экране смогут насладиться высококачественными изображениями, однако, если вы предложите такой же размер и качество посетителям мобильной версии, они, вероятнее всего будут неприятно удив-

[1] https://developer.mozilla.org/en-US/docs/Web/HTML/Supported_media_formats.
[2] https://caniuse.com/#feat=webp.
[3] https://w3techs.com/technologies/overview/image_format/all.
[4] https://tinypng.com/.

лены медленной загрузкой страниц. На рис. 6.6 показано, что размеры сайтов для настольных компьютеров и мобильных устройств постепенно уравниваются. Возможно, так происходит потому, что все больше сайтов переходит на адаптивный дизайн, подходящий для обеих версий, но используют разные методы доставки изображений разных размеров на две платформы.

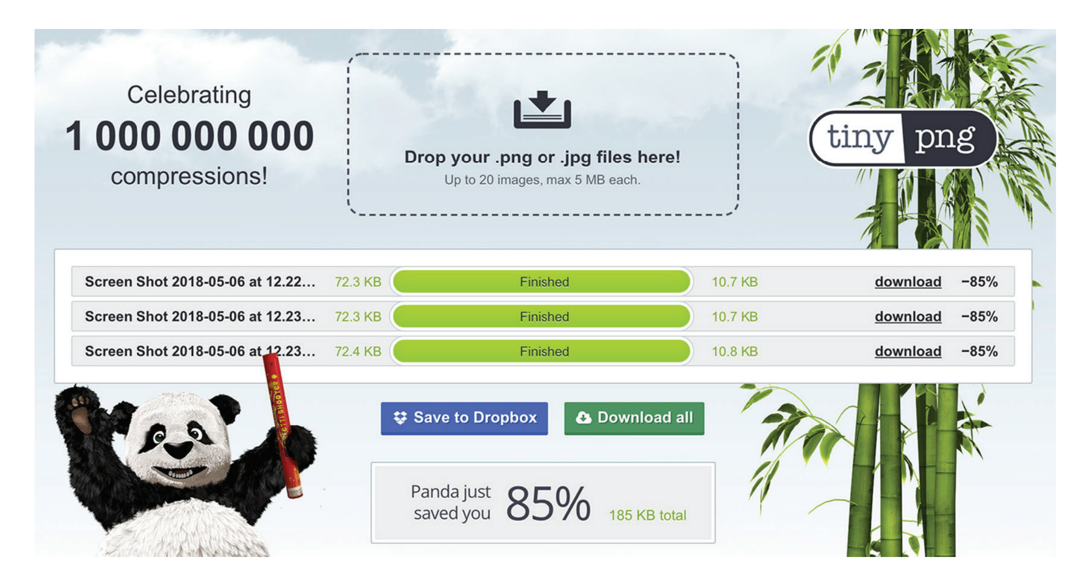


Рис. 6.5 **Tinypng может значительно уменьшить размер файла**

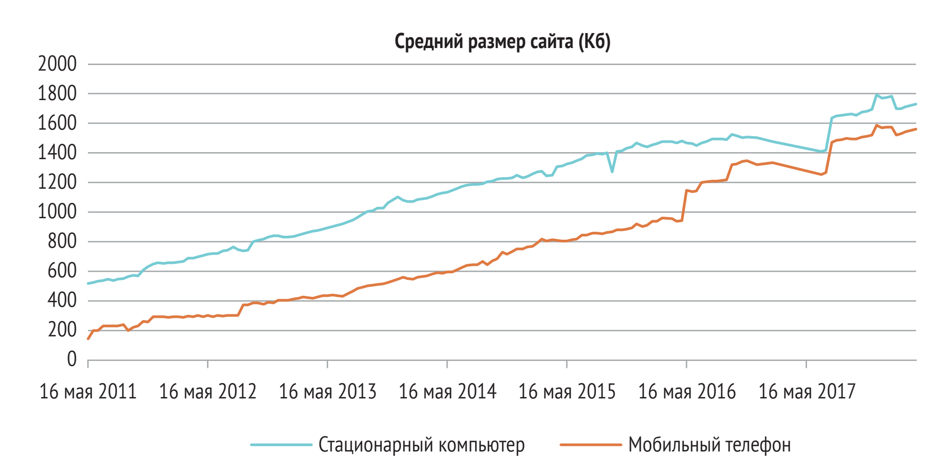


Рис. 6.6 **Размеры сайтов для мобильных и настольных устройств постепенно уравниваются**

Словом, HTTP/2 не меняет формат отправляемых файлов и передаваемых (или используемых) на стороне клиента данных. Используйте

наиболее подходящий вам формат файла и оптимизируйте медиафайлы, прежде чем разместить их на своем веб-сайте.

Сжатие текстовых данных

Несмотря на то что сжатие используется преимущественно для изображений, фундаментальные технологии Internet – HTML, CSS и JavaScript – основаны на тексте, и размер таких ресурсов также следует сокращать (настолько, насколько это возможно). В HTTP/1.1 тела HTTP сжимали с помощью gzip или аналогичных инструментов, что позволяло уменьшить объем отправляемых данных. В HTTP/2 вы можете придерживаться того же принципа. Ведь от версии протокола не меняется формат отправляемых данных, а *меняется только метод их отправки*. Таким образом, сжатие текстовых ответов в HTTP/2 так же актуально, как и в HTTP/1.1. Текстовые файлы легко поддаются сжатию, и вы можете уменьшить их объем до 90 %. В табл. 6.3 представлены коэффициенты сжатия распространенных библиотек JavaScript и CSS[1].

Таблица 6.3 Степень сжатия общих библиотек с использованием gzip

Библиотека	Размер исходного файла	Размер сжатого файла	Степень сжатия
jquery-1.11.0.js	276 Кб	82 Кб	70 %
angular-1.2.15.js	729 Кб	182 Кб	75 %
bootstrap-3.1.1.css	118 Кб	18 Кб	85 %
foundation-5.css	186 Кб	22 Кб	88 %

Единственным недостатком концепции сжатия данных перед их отправкой является то, что для сжатия на стороне сервера и распаковки на стороне браузера нужны время и вычислительные ресурсы, но для современного оборудования это мизерные затраты. На сетевом уровне преимущества отправки меньшего количества байтов почти всегда намного выше, чем затраты времени и вычислительной мощности на сжатие.

Самой распространенной технологией сжатия на сегодня остается gzip[2], однако популярность набирают и другие, такие как Brotli[3]. Иногда Brotli обеспечивает даже лучшие показатели (в зависимости от настроек[4]) и, следовательно, еще большую экономию ресурсов. Подобные ему инструменты можно использовать в дополнение к gzip для браузеров, которые еще не поддерживают Brotli[5]. HTTP/2 обрабатывает различные кодировки контента так же, как HTTP/1.1, поэтому переход на HTTP/2 не отменяет необходимости сжатия тел ответов и использования под-

1 https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimizeencoding-and-transfer?hl=en#text-compression-with-gzip.

2 https://w3techs.com/technologies/details/ce-compression/all/all.

3 https://opensource.googleblog.com/2015/09/introducing-brotli-new-compression.html.

4 https://blogs.akamai.com/2016/02/understanding-brotlis-potential.html.

5 https://caniuse.com/#feat=brotli.

ходящего формата (это важно!). Единственное незначительное отличие состоит в том, что для некоторых новых форматов, таких как Brotli, требуется HTTPS и, возможно, более поздние версии вашего веб-сервера. Если вы перешли на HTTPS и HTTP/2 одновременно или вам пришлось обновить программное обеспечение вашего веб-сервера, у вас есть возможность использовать новые форматы, хотя, возможно, лучше всего сначала выполнить переход и только потом применять новые форматы в сочетании со старыми.

Одним из главных преимуществ сжатия является то, что, за исключением необходимости небольшой настройки сервера, при его использовании не возникает других проблем. При правильной настройке веб-серверы сживают ресурсы буквально на лету, веб-браузеры автоматически распаковывают ресурсы, и большинство владельцев веб-сайтов может не думать о сжатии после его включения. Некоторые веб-серверы позволяют обслуживать предварительно сжатый контент, что снижает нагрузку на веб-сервер. Такой метод требует дополнительных усилий при добавлении нового контента (чтобы предварительно сжать его перед загрузкой), однако того же требует не только HTTP/1, но и HTTP/2.

Независимо от используемой версии HTTP, вам необходимо использовать сжатие тел HTTP-сообщений. Кодировка содержимого передается браузеру в HTTP-заголовке `content-encoding`, как это было в HTTP/1.1.

Сжатие HTTP-заголовков

Следует поговорить еще об одной вещи, которая действительно меняется при переходе на HTTP/2, – это сжатие заголовков. В HTTP/1 возможно сжимать только *тела запросов и ответов*, а HTTP/2 позволяет проделывать то же самое и *с заголовками*, используя при этом формат HPACK. В том числе с помощью него HTTP/2 снижает расходы производительности при выполнении нескольких запросов одновременно. Без HPACK отправка одного и того же содержимого по двум запросам приведет к увеличению количества данных заголовка в два раза. На самом деле такое нововведение очень важно, поскольку использование HTTP заголовков продолжает расти, а для многих небольших запросов заголовки могут составлять пропорционально большую часть как запроса, так и ответа.

Поскольку сжатие заголовков осуществляется непосредственно браузером и владельцам веб-сайтов и разработчикам не требуется прикладывать для этого много усилий, сейчас я не буду останавливаться на этой теме, однако в главе 8 мы все же это обсудим.

Минификация кода

Минификация кода является еще одним способом сокращения объема данных, будь то код HTML, CSS или JavaScript. Она заключается в удалении пробелов и комментариев из кода, и иногда даже его в переписывании в целях уменьшения размеров имен локальных переменных и удаления ненужных разделителей. Подобно сжатию, минификация со-

хранила актуальность в HTTP/2, поэтому при переходе на новую версию протокола я настоятельно рекомендую вам ее использовать.

Потенциально HTTP/2 позволяет применять код и без конкатенации строк (или, по крайней мере, сократить ее применение), поэтому этап окончательной сборки кода теперь может быть пропущен. Возможно, вы захотите использовать эту возможность, чтобы привести развернутый код в большее соответствие с исходным кодом, ведь необходимости в их различии уже нет. В таком случае, скорее всего, вы захотите отказаться от минификации кода. Отказ от нее может немного снизить производительность, но, на мой взгляд, минификация поверх сжатия только зря повышает ресурсоемкость, поскольку сжатие само по себе эффективно удаляет повторяющиеся символы (например, пробелы). С другой стороны, минифицированный код труднее читать, и такой вариант доставит трудности, если вы пытаетесь устранить проблемы в производственной среде. К сожалению, в средах разработки далеко не все ошибки легко поддаются устранению. Вы можете добавить карты кода, которые позволят вам сократить негативное влияние минификации на ваш рабочий код, однако в ходе этого процесса не исключено возникновение некоторых сложностей[1].

Рассмотрим преимущества минификации на реальном примере: популярная библиотека Bootstrap v4.0.0 имеет две версии кода CSS и JavaScript. Сравним минифицированные и оригинальные версии. Для этого откройте инструменты разработчика в Chrome, найдите столбец Size, а затем загрузите файл Bootstrap CSS с https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css, как показано на рис. 6.7.

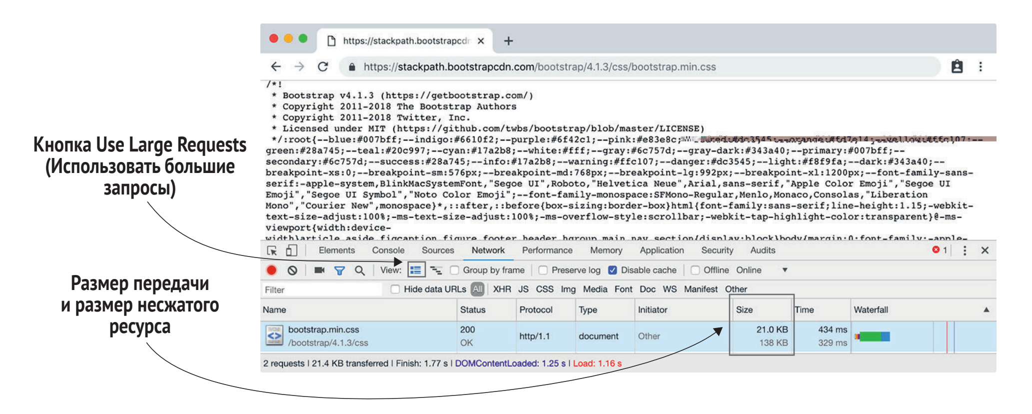


Рис. 6.7 Просмотр размеров сжатых и несжатых запросов

Если в столбце Size не отображаются оба варианта, нажмите кнопку Use Large Request Rows. В данном примере в Size показаны два значения: размер сжатого (21,0 Кб) и несжатого (138 Кб) кода. Однако технически эти числа несопоставимы: первое значение – это размер переданного

[1] https://www.html5rocks.com/en/tutorials/developertools/sourcemaps/.

сообщения, поэтому оно включает в себя HTTP-заголовки (запроса и ответа), а второе –это исходный размер отправляемого обратно тела. Сами заголовки имели размер примерно 0,5 Кб (без сжатия). Таким образом, различия имеют лишь незначительный эффект и могут быть проигнорированы. Если вы загрузите и сожмете файлы по отдельности (как в разделе 6.2.3), значение будет более точным. Однако здесь необходимо знать, какие настройки gzip используются в командной строке и веб-сервере.

В табл. 6.4 приведены результаты для неминифицированной версии.

Таблица 6.4 Сжатие gzip и минификация CSS-файла Bootstrap (v4.1.3)

Тип сжатия	Bootstrap.css	Bootstrap.min.css
Исходный файл	170 Кб	138 Кб (81 % от размера исходного файла)
gzip	22,8 Кб (13 % от размера исходного файла)	21,0 Кб (12 % от размера исходного файла)

Как видите, неминифицированная версия сжимается до 13 % от размера исходного файла, минифицированная версия – до 12 %. Мы видим разницу, но она не велика, всего 1 %, или 1,8 Кб. Безусловно, для таких популярных инструментов, как Bootstrap, на счету каждый байт, и здесь имеет смысл предоставить пользователям уменьшенную версию библиотеки. Однако для вашего кода экономия может быть не столь значительной.

Экономия зависит от конкретного кода. Например, если для CSS-файла Bootstrap сжатие не дало значительного результата, то файл JavaScript того же Bootstrap действительно уменьшился в объеме (см. табл. 6.5).

Таблица 6.5 Сжатие gzip и минификация файла JavaScript Bootstrap (v4.1.3)

Тип сжатия	Bootstrap.js	Bootstrap.min.js
Исходный файл	121 Кб	49,8 Кб (41 % от размера исходного файла)
gzip	20,9 Кб (17 % от размера исходного файла)	14,2 Кб (12 % от размера исходного файла)

Здесь мы наблюдаем большую степень сжатия (полученный в результате файл составляет 17 % от исходного только с помощью gzip, 12 % с помощью gzip и минификации) и, соответственно, большую эффективность: экономия составляет 5 %, или 6,7 Кб. Зачастую JavaScript содержит больше комментариев к коду, пробелов и имен переменных, чем CSS; отсюда и полученные нами результаты. Таким образом, за счет минификации (без сжатия) размер файла JavaScript сокращается до 41 %, а размер CSS составляет 81 % от исходного размера. К слову, минифицированные файлы намного больше файлов, сжатых с помощью gzip, что подтверждает мою точку зрения о том, что gzip (или аналогичный тип сжатия) является лучшим методом оптимизации; конечно, минификацию можно использовать, ведь она тоже дает дополнительные преимущества.

Минификация намного полезнее для крупных сайтов, владельцы которых обладают необходимым опытом. В табл. 6.6, как и в 6.5, приведены примеры сжатия кода распространенных библиотек веб-разработки. Особенно сильно разница заметна в минифицированных версиях некоторых библиотек, в частности jQuery и Angular.

Таблица 6.6 Степень сжатия некоторых распространенных библиотек

Библиотека	Размер исходного файла	Размер сжатого файла	Степень сжатия
jquery-3.3.1.js	265 Кб	78,9 Кб	30 %
jquery-3.3.1.min.js	84,9 Кб	30,0 Кб	35 %
angular-1.7.2.js	960 Кб	297 Кб	31 %
angular-1.7.2.min.js	168 Кб	56,6 Кб	34 %
bootstrap-4.1.3.css	121 Кб	20,9 Кб	17 %
bootstrap-4.1.3.min.css	49,8 Кб	14,2 Кб	29 %
foundation-6.4.3.css	158 Кб	18,8 Кб	12 %
foundation-6.4.3.min.css	118 Кб	14,7 Кб	12 %

Обфускация кода – это еще одна причина минификации. Однако попытка использовать обфускацию с целью затруднения анализа кода нецелесообразна, так как неминифицированный код довольно тривиален. Тем не менее удаление комментариев может предотвратить утечку каких-либо неловких вставок от разработчиков (например, комментарии вроде «Когда-нибудь я исправлю этот ужасный код»). В теории минифицированный код помогает браузеру работать быстрее и эффективнее, хотя уже на первом шаге любого синтаксического анализа код и так минифицируется, поэтому значительных преимуществ вы можете и не получить.

Подводя итог, можно сказать, что решение применять минифицацию кода или нет не зависит напрямую от HTTP/2. Однако, если после перехода на HTTP/2 изменились методы разработки (например, вы стали меньше применять конкатенацию), возможно, стоит пересмотреть отношение к минификации кода и проследить, актуально ли ее использование в данный момент. Минификация сложнее, чем сжатие файлов веб-сервером, а при их совместном использовании ее эффективность становится намного меньше.

6.3.2 Предотвращение повторной отправки данных с помощью кеширования

«Самый быстрый HTTP-запрос – тот, который не был сделан», так гласит самая известная цитата о веб-производительности (хотя, я думаю, никто не помнит, кто сказал это первым[1]). Целью HTTP/2 является улучшение производительности отправки HTTP-запросов. Применение кеширования (настолько часто, насколько это возможно) поможет значительно ускорить данный процесс. В этом случае, если пользователям понадобится какой-либо ресурс, уже используемый ранее, они смогут взять его из кеша HTTP, что намного быстрее, чем выполнение полного сетевого запроса, независимо от того, используете ли вы протокол HTTP/1, HTTP/2, или какую-либо будущую версию.

---

[1] Наиболее вероятный кандидат, Стив Содерс, отрицает, что это был он, несмотря на то что цитата часто приписывается именно ему: https://www.stevesouders.com/blog/2012/03/22/cache-them-if-you-can/.

В HTTP/1 необходимо было использовать кеширование настолько, насколько это представлялось возможным. Принцип остается таковым и для HTTP/2. Актуальны и HTTP-заголовки `cache-control` и `expires` (хотя некоторые люди небезосновательно утверждают, что заголовок `expires` утратил актуальность, ввиду того, что почти все клиенты сейчас понимают HTTP/1.1, а HTTP/1.0 используется крайне редко, особенно теми приложениями, которым необходимо кеширование[1].

Кеширование может быть сложным. Рациональное и постоянное использование кеширования – это сложная тема, требующая размышлений и использования методов очистки кеша[2]. Чаще всего сайты, использующие кеширование общих ресурсов (таблиц стилей, JavaScript, логотипов и т. д.) более быстрые и отзывчивые, а те, которые его не используют, загружаются медленно, вследствие чего становятся неудобными в использовании. Поскольку кеширование является важной темой, необходимо разобраться в HTTP-кешировании и том, как работает код ответа `304` (Not Modified), о котором я кратко упомянул в главе 1.

Полученный HTTP-ответ может включать в себя HTTP-заголовок `cache-control` (или устаревший `expires`), который указывает на то, как долго ресурс должен считаться действительным. Рассмотрим пример из реальной практики. Если вы загрузите Wikipedia в браузере, вы увидите заголовок, как на рис. 6.8.

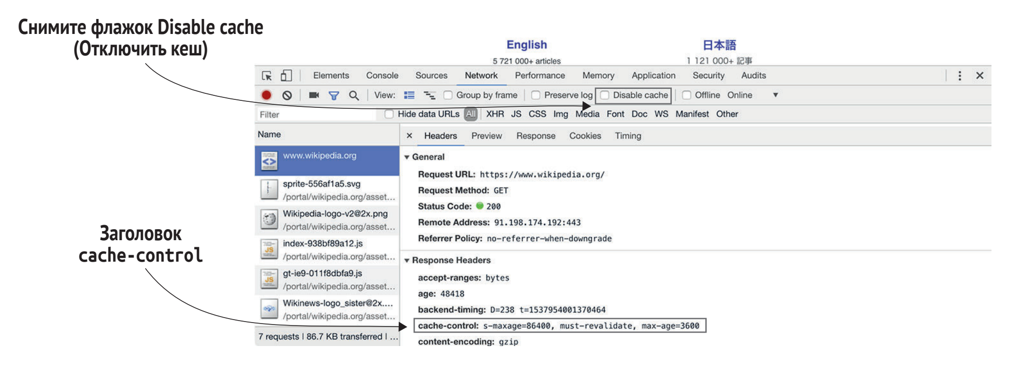


Рис. 6.8 Заголовок `cache-control` в Wikipedia

На этом рисунке показано, что домашнюю страницу Wikipedia можно кешировать в течение 3600 с (`max-age=3600`), или 1 ч, после чего ее необходимо повторно проверить перед использованием (`mustrevalidate`). Также здесь указано, что промежуточные кеши, такие как прокси, могут кешировать страницу в течение 86 400 с, или 1 дня (`s-maxage=86400`). Однако зачастую для поддержки своей актуальности (тема, выходящая за рамки этой книги) они используют другие методы, поэтому в этом плане данную настройку можно проигнорировать.

1 https://www.fastly.com/blog/headers-we-dont-want.
2 https://css-tricks.com/strategies-for-cache-busting-css/.

Убедитесь, что флажок **Отключить кеш**, показанный на рисунке, снят. Затем пролистайте страницу назад и вперед, после чего вы увидите что-то похожее на рис. 6.9.

Name	Status	Protocol	Type	Initiator	Size
www.wikipedia.org	200	h2	document	Other	(from disk cache)
sprite-556af1a5.svg /portal/wikipedia.org/assets/img	200	h2	svg+xml	(index) Parser	(from disk cache)
Wikipedia-logo-v2@2x.png /portal/wikipedia.org/assets/img	200	h2	png	(index) Parser	(from disk cache)
index-938bf89a12.js /portal/wikipedia.org/assets/js	200	h2	script	(index) Parser	(from disk cache)
gt-ie9-011f8dbfa9.js /portal/wikipedia.org/assets/js	200	h2	script	(index) Parser	(from disk cache)
Wikinews-logo_sister@2x.png /portal/wikipedia.org/assets/img	200	h2	png	(index) Parser	(from disk cache)

Рис. 6.9 Wikipedia загружается из дискового кеша

Как и ожидалось, веб-сайт был загружен из кеша диска, и это отражено в столбце Size. Если вы видите надпись `from memory cache`, а не `from disk cache`, вероятно, вы перешли с другой страницы Wikipedia, а не с другого сайта, и поэтому эти ресурсы находятся в свежем кеше памяти, однако принцип остается неизменным. На рис. 6.9, также представлен старый кешированный ответ с кодом состояния `200` (выделен темным цветом, чтобы показать, что это кешированный ответ, который трудно увидеть на рисунке).

Ради интереса подождите час, пока истечет срок действия кеша, а затем повторите эксперимент. Чтобы сэкономить время, перезагрузите страницу (что даст тот же эффект). Вы увидите что-то похожее на рис. 6.10.

Name	Status	Protocol	Type	Initiator	Size
www.wikipedia.org	304	h2	document	Other	560 B 74.0 KB
sprite-556af1a5.svg /portal/wikipedia.org/assets/img	200	h2	svg+xml	(index) Parser	(from memory c...
Wikipedia-logo-v2@2x.png /portal/wikipedia.org/assets/img	200	h2	png	(index) Parser	(from memory c...
index-938bf89a12.js /portal/wikipedia.org/assets/js	200	h2	script	(index) Parser	(from memory c...
gt-ie9-011f8dbfa9.js /portal/wikipedia.org/assets/js	200	h2	script	(index) Parser	(from memory c...
Wikinews-logo_sister@2x.png /portal/wikipedia.org/assets/img	200	h2	png	(index) Parser	(from memory c...

Рис. 6.10 Перезагрузка Wikipedia из устаревшего кеша

На этом рисунке вместо кода ответа `200` мы видим код `304`. Если вы перезагрузите это окно, вы также увидите, что Chrome использовал кеш

изображений в памяти для изображений, а не HTTP-кеш диска, как было на рис. 6.9. Ответ 304 появляется потому, что браузер выполнил условный запрос GET, как показано на рис. 6.11.

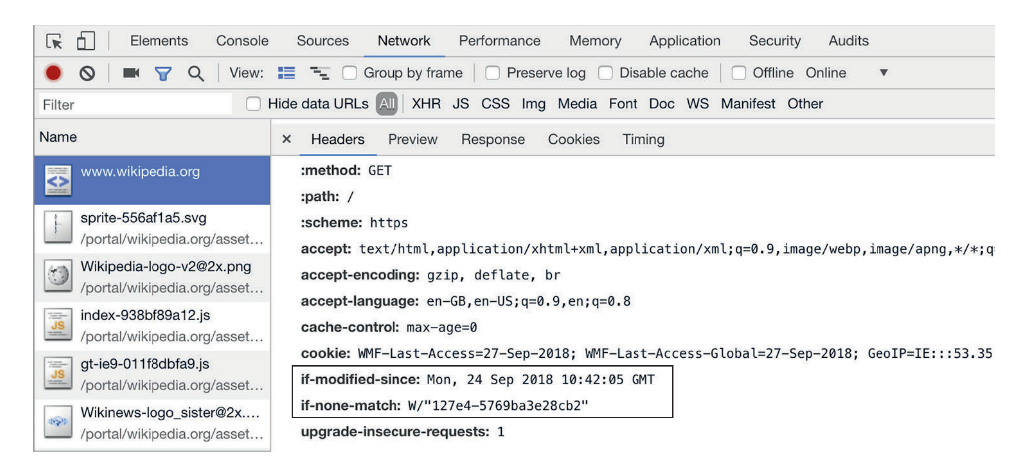


Рис. 6.11 Условный запрос GET

Браузер нашел домашнюю страницу в кеше, обнаружил, что она устарела, и отправил новый запрос: «Отправьте мне новую версию страницы (`if-modified-Since`) или значение `eTag` (`if-none-match`), которое вы отправляли со страницей в последний раз». Значение `eTag` позволяет проанализировать актуальность кешированной страницы не только по дате. Оно зависит от реализации и может быть, например, хешем содержимого. Если указаны оба значения (как на рис. 6.11), приоритет имеет значение `eTag`, указанное в заголовке `if-none-match`. Сервер совершает проверку и видит, что страница не изменилась, затем отправляет ответ 304, чтобы сказать, что копия, имеющаяся в веб-браузере, все еще актуальна. Ответ 304 не имеет тела, поэтому он загружается быстрее, чем полноценный ресурс.

Ответы 304 так и остались медленными; на протяжении всего своего пути они будут требовать сетевых запросов. В HTTP/2 цикл прием-передачи стал менее затратным (но не полностью!). Однако при использовании HTTP/1 затраты на отправку ответа 304 были почти такие же, как отправка полного ответа 200, поэтому ответы 304, возможно, использовались не так часто, ведь HTTP-запросы были относительно ресурсоемкими. Многие веб-сайты вообще не кешируют свои HTML-страницы. В таких случаях каждый раз, когда вы переходите на домашнюю страницу, браузер заново загружает ее, а затем использует кеш только для доступа к необходимым ему ресурсам. Отсутствие кеширования ресурсов веб-страницы может замедлить просмотр веб-сайта. Если вы находитесь на домашней странице и нажимаете на другую страницу, а затем хотите вернуться обратно на домашнюю страницу, перезагрузка должна быть мгновенной, но часто случается небольшая задержка. Директива `cache control` на всех веб-страницах сайта сделает его более отзывчивым (по-

скольку они будут загружаться из кеша), а также поможет сэкономить пропускную способность (за счет использования ответов `304` даже после истечения срока действия кеша).

Важно понимать, что отправка ответов `304` по-прежнему требует затрат, поэтому я не предлагаю вам использовать их как замену кешированию. Однако можно применять их как замену некоторых ресурсов, которые вы не кешировали. Когда я делал скриншоты для предыдущих рисунков, я изо всех сил пытался найти веб-сайт, который кешировал бы саму веб-страницу, как это делает Wikipedia. Новостным сайтам и социальным сетям очень важно выдавать пользователю только актуальные страницы. Для этого есть и другие, более эффективные способы загрузки контента на страницу, чем использование предварительно сгенерированного HTML, который не следует кешировать с помощью JavaScript AJAX. Я считаю, что веб-сайты должны кешировать страницы на недолгий период, по крайней мере при использовании HTTP/2, потому что ответ `304` менее затратен, чем ответ `200`.

Согласно многим рекомендациям по HTTP-кешированию, необходимо использовать именно долговременное кеширование, чтобы сэкономить ресурсы при последующих входах на сайт. Данный аспект, возможно, не так важен, как использование кеширования для просмотра сайта в текущем сеансе, где сайт будет отзывчивым для пользователя. Использование кеширования для оптимизации просмотра сайта непосредственно во время сеанса хорошо работает и с краткосрочным кешированием. Кроме того, краткосрочные кеши помогут избежать частого проведения процедуры очистки кеша. Точно так же использование кеширования на стороне сервера для предотвращения запросов пограничного сервера к внутреннему серверу может привести к значительным улучшениям, даже при краткосрочном кешировании[1].

Одним из других недостатков краткосрочного кеширования является то, что ресурсы могут быть удалены из кеша, ввиду того что браузер счел их недействительными. Лучше «использовать долгосрочное кеширование и время от времени перепроверять и подтверждать актуальность ресурсов», но такой возможности у нас нет. Марк Ноттингем (Mark Nottingham), сопредседатель рабочей группы HTTP[2], предложил использовать опцию `Stale-While-Revalidate3`, которая делает такой сценарий возможным, однако пока что ни один браузер не воспользовался этим предложением.

В заключение стоит сказать, что HTTP/2 не меняет параметры кеширования напрямую, но снижение затрат на осуществление сетевых запросов может привести к переоценке стратегий кеширования. Кроме того, было предложено внести изменения в HTTP/2 push, которые позволили бы обновлять кеши (см. главу 5), однако в настоящее время осуществить такое невозможно.

---

1 https://www.nginx.com/blog/benefits-of-microcaching-nginx/.

2 https://httpwg.org/.

3 https://www.mnot.net/blog/2014/06/01/chrome_and_stale-while-revalidate.

6.3.3 Снижение нагрузки на сеть посредством сервис-воркеров

Сервис-воркеры[1] (Сценарии, выполняемые в фоновом режиме в браузере пользователя. – *Прим. ред.*) появились относительно недавно, но они уже доступны во всех современных браузерах, кроме Internet Explorer 11[2]. Они являются одним из способов запуска прокси-сервера JavaScript, который находится между веб-страницей и сетью, как показано на рис. 6.12.

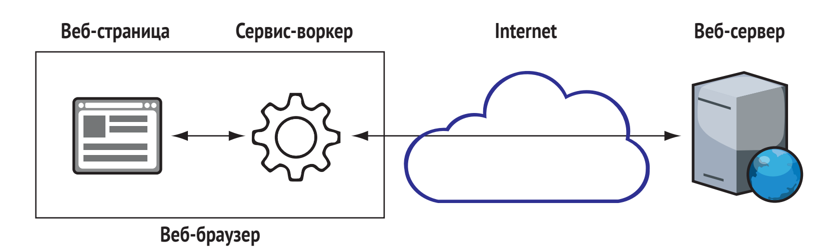


Рис. 6.12 Сервис-воркеры

Сервис-воркеры могут просматривать и менять HTTP-запросы, а также отвечать на них. Также с их помощью можно обеспечить аналогичные возможности в нативных мобильных приложениях, в том числе в автономном режиме. Даже если кешируется сама веб-страница, при перезагрузке она будет пытаться подключиться к веб-сайту, чтобы проверить, действительна ли кешированная версия и перезагрузка не удастся, если нет сетевого подключения. В мобильных приложениях, которые не позволяют производить обновление в офлайн-режиме, такого обычно не происходит. Если сервис-воркер используется на веб-сайте, он может прерывать запросы и в автономном режиме возвращать ранее кешированную версию ресурса. Данный метод позволяет кешированному сайту загружаться даже в автономном режиме, как в мобильных приложениях.

Сервис-воркеры предоставляют много интересных возможностей для оптимизации HTTP на стороне веб-разработки. Вы можете использовать кратковременное кеширование, но при этом не удалять элементы из кеша сервис-воркера даже после истечения срока его актуальности. Они позволят вам использовать ответ `304` чаще и не прибегать к долгосрочному кешированию. Использование сервис-воркеров не меняется при переходе на HTTP/2, и вообще, им стоит посвятить отдельную книгу, поэтому я не думаю, что сейчас нужно рассказывать о них что-то еще. Однако мне бы очень хотелось, чтобы в течении следующих нескольких лет они стали более популярными, поскольку сервис-воркеры предоставляют мощные возможности для обработки HTTP-запросов.

---

1 https://developers.google.com/web/fundamentals/primers/service-workers/.

2 https://caniuse.com/#feat=serviceworkers.

6.3.4 Не отправляйте то, что вам не нужно

Продолжая тему отказа от ненужных HTTP-запросов, отмечу, что еще одним способом оптимизации производительности улучшением является отправка только действительно необходимых данных. Хотя этот момент может показаться очевидным, существует множество причин, ввиду которых вы можете отправлять данные, которые на самом деле не используются.

Методы, упомянутые в разделе 6.2 – объединение файлов и спрайтинг изображений, – зачастую инициируют отправку большего количества данных, что может быть остро необходимо для уменьшения количества HTTP-запросов. Поскольку в HTTP/2 необходимость в сокращении количества запросов меньше, вы можете пересмотреть эти методы с точки зрения уменьшения объема данных. Возможно, вам будет удобнее продолжать использовать их, если вы интегрировали их в процесс сборки версий веб-сайтов и, следовательно, не видите острой необходимости их удалять. Однако, возможно, используя эту технику, вы отправляете больше файлов, чем нужно на самом деле.

Ненужные ресурсы могут загружаться и во время многих других процессов. Например, существует вероятность, что вы загружаете изображения, которые не отображаются в мобильной версии, но на самом деле равно присутствуют на веб-сайте. HTTP/2 не вносит ограничение на отправку ненужных ресурсов. Фактически он лишь добавляет новый способ отправки (HTTP/2 push), который требует особого внимания, как обсуждалось в главе 5.

Может казаться, что в HTTP/2 загрузка ресурсов происходит очень быстро, но на самом деле этот протокол никак не решает проблему отправки большого количества данных (не считая сжатия HTTP-заголовков). Именно поэтому сайтам необходимо продолжать контролировать загрузку данных и загружать только самое необходимое.

6.3.5 Подсказки для ресурсов HTTP

В главе 5 мы говорили о подсказке, предлагающей серверу осуществить предварительную загрузку. Она является лишь одной из многих подобных подсказок[1], касающихся ресурсов, которые могут быть полезны для дальнейшей оптимизации использования HTTP. Данные подсказки актуальны как для HTTP/2, так и для HTTP/1. Для HTTP они реализованы в виде заголовков, а для HTML в виде элементов `<link>`. Несмотря на то что подсказки существуют уже довольно давно, поддержку они получили лишь недавно. Они представляют собой новые способы дополнить HTTP/2.

Предварительная выборка DNS

Данная подсказка использовалась сайтом Amazon, как показано в примерах из главы 2. Она находится в разделе `HEAD` кода домашней страницы:

---

[1] https://w3c.github.io/resource-hints/.

```
<link rel='dns-prefetch' href='//m.media-amazon.com'>
```

Как следует из самого названия подсказки, она побуждает сервер выполнять поиск соединения в DNS еще до того, как оно потребуется, что значительно экономит время (см. строку 17 на рис. 6.13).

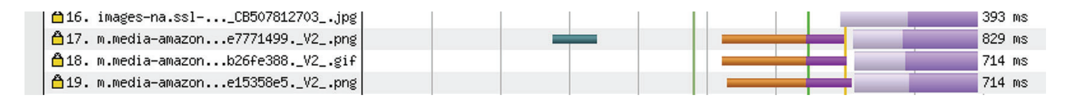


Рис. 6.13 Предварительная выборка DNS на практике

Пусть предварительная выборка DNS и экономит совсем немного времени, но зато для ее реализации требуется меньше строк кода. Также ее поддерживают все основные браузеры (однако время поддержки ограничено)[1]. DNS-запросы имеют ограниченное время жизни (*TTL* – *Time to Live*), поэтому иногда веб-сайтам не следует искать домен раньше времени (например, тот, который будет использоваться исключительно для дальнейшего перехода на другие страницы веб-сайта), так как поиск может повториться, если TTL истечет. Обычно TTL составляет 300 или больше секунд, и в идеале загрузка ваших страниц занимает не более 5 мин, поэтому использовать подсказки для текущей страницы безопасно. Выполнять предварительную выборку DNS перед ссылкой на ресурс нет смысла, так как эта ссылка все равно будет выполнять поиск DNS, поэтому подсказки следует выполнять только для ресурсов, обнаруженных позднее. Данный метод наиболее полезен для соединения, необходимого зависимому ресурсу, который не будет отображен после синтаксического анализа HTML. Большинство веб-сайтов загружает контент из разных доменов, поэтому использование этого заголовка обеспечивает хорошие результаты.

Предварительная установка соединения

Предварительная установка соединения продвигает концепцию предыдущей подсказки на один шаг вперед. Она не только выполняет поиск DNS, но и устанавливает соединение, что может сэкономить затраты на согласование TCP и HTTPS, связанные с установкой новых подключений. Данную подсказку поддерживает большинство современных браузеров[2]. Однако ее следует использовать в определенное время; если вы воспользуетесь ей слишком рано, алгоритм медленного запуска TCP сработает после периода бездействия, или, что еще хуже, соединение будет разорвано (см. главу 9).

Как и предварительная выборка DNS, предварительная установка соединения полезна при загрузке критических ресурсов из других доменов.

---

[1] https://caniuse.com/#feat=link-rel-dns-prefetch.

[2] https://caniuse.com/#feat=link-rel-preconnect.

Предварительная выборка

Предварительная выборка осуществляет запрос ресурсов с низким приоритетом. В отличие от предварительной загрузки (которую мы обсудим далее), которая нацелена на ускорение загрузки текущей страницы, предварительная выборка обычно используется для последующих переходов по веб-страницам; поскольку ресурсы получены с низким приоритетом, они не будут использоваться, пока не загрузится текущая страница. Загружаемые с помощью этой подсказки ресурсы хранятся в кеше и уже готовы к использованию. Подсказка поддерживается большинством современных браузеров[1], кроме Safari (на момент написания этой книги). Я считаю, что из-за HTTP/2 в подсказку предварительной выборки не будет внесено каких-либо изменений.

Предварительная загрузка

Метод предварительной загрузки подсказывает браузеру загрузить ресурс для конкретной страницы с высоким приоритетом. Данная подсказка является следующим локальным шагом после предварительной установки соединения, но, в отличие от нее, он предназначен именно для ресурсов на странице. Веб-браузеры довольно хорошо умеют сканировать HTML и загружать все необходимые ресурсы, но предварительная загрузка позволяет загрузить ресурсы, которые не включены непосредственно на страницу (например, шрифты, указанные в файлах CSS) заранее. Распространение данной подсказки заняло долгое время, однако сейчас ее поддерживает большинство современных браузеров[2].

Особенно актуальна предварительная загрузка для HTTP/2 push (как обсуждалось в главе 5). По большей части так произошло, потому что данная подсказка была создана именно для этой цели, а не выступала предлагаемым вариантом использования. Учитывая сложность HTTP/2 push, подсказка предварительной загрузки (без HTTP/2 push) может оказаться наиболее простым вариантом. Сегодня многие люди рекомендуют использовать предварительную загрузку вместо HTTP/2 push, и в этом случае обязательно используйте атрибут `nopush` при использовании заголовка ссылки (нет необходимости использовать его с версией HTML, поскольку веб-серверы не используют его как указание отправить ресурс).

Предварительная загрузка может принести еще большую пользу, если код HTTP-ответа `103 Early Hints` (обсуждаемый в главе 5) станет более распространенным, поскольку он может содержать заголовки ссылок предварительной загрузки HTTP (даже в HTTP/1.1).

Предварительная визуализация

Предварительная визуализация является самой затратной подсказкой. Она позволяет загружать и выполнять предварительную визуализацию

---

1 https://caniuse.com/#feat=link-rel-prefetch.
2 https://caniuse.com/#feat=link-rel-preload.

целых страниц (включая все необходимые ресурсы). Ключевой идеей этой подсказки является то, что, если можно надежно оценить навигацию по следующей странице, страница может быть загружена мгновенно. На момент написания этой книги предварительную визуализацию поддерживают только Chrome и IE 11[1], хотя Chrome стремится отказаться от использования данной подсказки и, возможно, скоро сделает это[2]. Риск чрезмерного использования предварительной визуализации довольно высок, что приводит к трате пропускной способности и увеличивает время обработки для клиента. Я считаю, что для HTTP/2 в подсказку предварительной визуализации не будут внесены изменения. Кроме того, вряд ли она станет приоритетной для реализации в других браузерах.

6.3.6 Сокращение задержки «последней мили»

Цель HTTP/2 – сократить влияние задержки за счет того, что во время выполнения HTTP-запроса TCP-соединение может использоваться и для других запросов. Однако HTTP/2 не решил проблему задержки, и нам все так же следует предпринимать различные меры для ее сокращения. Веб-серверы обычно подключены к Internet с помощью высокоскоростной инфраструктуры с широкой полосой пропускания, но посетители веб-страниц часто подключаются через менее надежные соединения, например через широкополосные и мобильные сети. «Последняя миля» – это канал, соединяющий конечное оборудование с инфраструктурой провайдера, и часто задержка здесь особенно сильно влияет на соединение.

Самый простой способ решить эту проблему – расположиться как можно ближе к браузеру, что для глобальных веб-сайтов обычно предполагает наличие сети локальных серверов рядом с вашей пользовательской базой. Такая сеть может быть локальной или (что встречается все чаще) сетью CDN. Большинство CDN поддерживает HTTP/2[3]. Учитывая сложности обновления вашего веб-сервера для его поддержки и более высокие требования HTTPS для него, CDN для вашего веб-сервера – простой вариант для перехода на HTTP/2 (как упоминалось в главе 3), а также гарантия сокращения задержки.

6.3.7 Оптимизация HTTPS

Во всем мире разработчики постепенно переходят на HTTPS. Новые функциональные возможности, такие как протокол HTTP/2, требуют поддержки HTTPS. Так происходит по ряду причин, как технических, так и идеологических, поскольку люди, разрабатывающие ключевые компоненты Internet (создатели браузеров, рабочая группа HTTP и т. д.), счита-

---

1 https://caniuse.com/#feat=link-rel-prerender.

2 https://groups.google.com/a/chromium.org/forum/#!topic/blink-dev/0nSxuuv9bBw.

3 http://cdncomparison.com/.

ют, что все в этом мире должно быть зашифровано. Изначально HTTPS использовался только определенными веб-сайтами или на определенных веб-страницах этих сайтов, но сейчас веб-сайты переходят на использование HTTPS на *всех* страницах.

Популярность HTTPS за последние несколько лет значительно возросла ввиду появления множества бесплатных центров сертификации, таких как Let's Encrypt[1]. Кроме того, HTTPS активно продвигают создатели браузеров Chrome[2] и Mozilla Firefox[3]. Динамика приведена на рис. 6.14[4]. Таким образом, HTTP/2 дает еще больше причин для перехода на такой тип соединений.

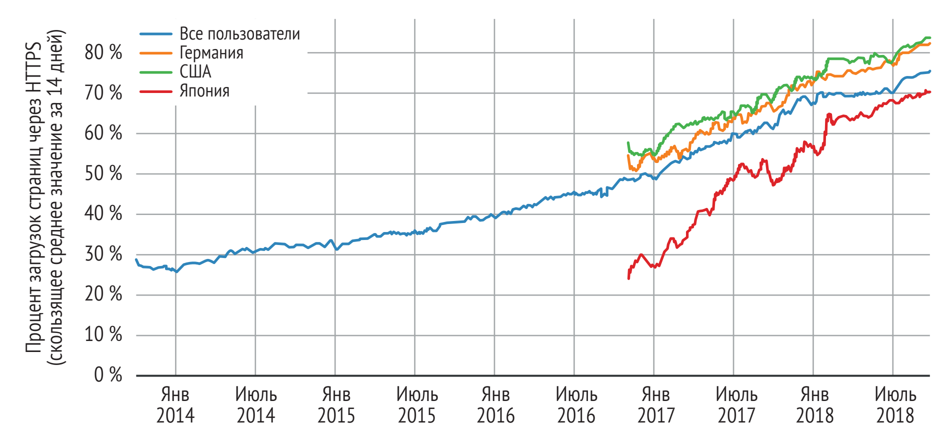


Рис. 6.14 Рост популярности HTTPS за последние несколько лет по данным Let's Encrypt (на основе статистики телеметрии Firefox)

Во время установки HTTP-соединения при загрузке веб-страниц возникают дополнительные задержки. После установки сеанса задержки минимальны, поскольку вычислительные ресурсы, необходимые для шифрования и дешифрования трафика, незначительны даже для мобильных устройств, однако это влияет на начальное время соединения. Чтобы попробовать подключиться заранее и уменьшить эффект для любых зависимых доменов, вы можете воспользоваться предварительной установкой соединения (эту подсказку мы обсуждали в разделе 6.3.5). Однако при первом подключении такой метод не сработает.

Для того чтобы сократить время, необходимое на установку HTTPS-соединения, обеспечить высокий уровень безопасности и предотвратить появление предупреждений от браузера, важно убедиться, что настройка HTTPS оптимизирована. Данный аспект важен для всех сайтов, исполь-

1 https://letsencrypt.org/.
2 https://blog.chromium.org/2018/02/a-secure-web-is-here-to-stay.html.
3 https://blog.mozilla.org/security/2017/01/20/communicating-the-dangers-of-non-secure-http/.
4 https://letsencrypt.org/stats/#percent-pageloads.

зующих HTTPS (и, следовательно, всех сайтов, использующих HTTP/2), поскольку он означает, что вы используете HTTPS из-за требований браузера. Ниже приведены рекомендации по обеспечению оптимального использования HTTPS (с HTTP/1 или HTTP/2):

- *во избежание предупреждений о смешанном содержимом убедитесь, что вы загружаете исключительно HTTPS-ресурсы*. В политике безопасности контента вы можете использовать конфигурацию `upgrade-insecure-requests`[1], поскольку она поддерживается большинством браузеров[2];
- *убедитесь, что ваш сертификат HTTPS вовремя обновляется*. В случае, если у вашего сертификата истечет срок действия, доступ к сайту будет закрыт. Раньше обновление выполнялось вручную, но сейчас благодаря Let's Encrypt процесс стал автоматическим. Дело в том, что Let's Encrypt допускает только 90-дневные кратковременные сертификаты, ввиду чего автоматизация просто неизбежна, ведь выполнение этого процесса вручную намного дольше и требует больших затрат;
- *регулярно проверяйте настройку HTTPS*. Протоколы HTTPS, шифры и конфигурация часто меняются, при этом добавляются новые параметры, а старые параметры становятся менее безопасными по мере увеличения вычислительной мощности. С помощью онлайн-инструмента SSLLabs Server Test[3] вы можете всесторонне протестировать настройки HTTPS, а также узнать про известные уязвимости и передовые методы в мире HTTPS. Оценка A гарантирует, что у вас нет проблем с HTTPS, а регулярное сканирование (не реже одного раза в квартал) для поддержания этой оценки позволит вам избежать неприятных сюрпризов;
- *реализуйте возобновление сеанса*[4] *защиты транспортного уровня* (*TLS*). Установка рукопожатия TLS требует значительного времени и усилий, поэтому его оптимизация имеет решающее значение. Один из лучших способов сделать это – разрешить *возобновление сеанса* TLS, тогда для каждого нового соединения не нужно будет устанавливать новое рукопожатие. В HTTP/2 следует использовать меньше соединений, но для более поздних или дополнительных соединений (например, сертифицированных и несертифицированных соединений) производительность может быть выше. Возобновление сеанса TLS может привести к некоторым проблемам с безопасностью[5], поскольку при последующем повторном подклю-

1 https://www.w3.org/TR/upgrade-insecure-requests/.
2 https://caniuse.com/#feat=upgradeinsecurerequests.
3 https://www.ssllabs.com/ssltest/.
4 https://calendar.perfplanet.com/2014/speeding-up-https-with-session-resumption/.
5 https://timtaubert.de/blog/2014/11/the-sad-state-of-server-side-tls-session-resumption-implementations/.

чении соединение HTTPS может быть слабее (хотя TLSv1.3 решает большинство этих проблем). Так или иначе большинство веб-сайтов по-прежнему стремится реализовать возобновление TLS с целью получить значительный прирост производительности;

- *не переусердствуйте с безопасностью*. Конечно, безопасность важна, но тем не менее всегда должен быть баланс безопасности и производительности. Если вы разрешите использовать только последний протокол TLS и шифрование с самыми надежными настройками, ваш веб-сайт будет крайне медленным, и более того, многие пользователи не смогут получить доступ к нему. На момент написания этой книги для обеспечения безопасности достаточно ключа RSA 2048, TLSv1.2 и TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256; кроме того, их поддерживает большинство браузеров. С течением времени ситуация изменится, но одно правило остается неизменным – выбирайте разумный уровень безопасности. Для создания соответствующей конфигурации для общих веб-серверов вы можете воспользоваться Mozilla Configuration Generator[1], а для сканирования настроек других сайтов инструментом SSLLabs. Я часто сканирую сайт ssllabs.com при помощи их собственного инструмента и думаю, что SSLLabs лучше всех знает, как его настроить;
- *подумайте, какая настройка HTTPS подходит вам лучше всего*. HTTPS – это все же очень сложно. Зачастую, вместо того чтобы управлять его настройкой самостоятельно, намного проще воспользоваться будет гарантировать пользователям стабильно безопасную настройку HTTPS. Однако ключи своей настройки HTTPS можно передавать только доверенной стороне; в противном случае весь смысл процесса теряется!
- *когда TLSv1.3 станет доступен, воспользуйтесь им*. Данный протокол был стандартизирован в августе 2018 года[2], но на данный момент он может быть недоступен для многих читателей. Протокол может обеспечить гораздо больший прирост производительности (и безопасности) по сравнению с предыдущими версиями[3].

Совсем скоро HTTPS войдет по всеобщее употребление. Его уже поддерживают многие сайты, и они могут пользоваться всеми приведенными выше советами. Одной из причин перехода большинства сайтов на HTTPS является HTTP/2, требующий использования только безопасных соединений. HTTPS требует настройки, решение относительно которой следует принимать непосредственно владельцам веб-сайтов. Любому создателю веб-сайта следует всерьез рассмотреть все упомянутые выше аспекты независимо от того, использует ли он HTTP/2.

---

1 https://mozilla.github.io/server-side-tls/ssl-config-generator/.

2 https://tools.ietf.org/html/rfc8446.

3 https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/48.

6.3.8 Методы повышения веб-производительности, не связанные с HTTP

В этом разделе мы говорим преимущественно о советах и методах, связанных с передачей ресурсов через HTTP. Однако многие другие способы улучшения веб-производительности не связаны с тем, как загружаются данные. В частности, замедлить прорисовку веб-сайта может неэффективный JavaScript. Также загрузка множества рекламных сетей и трекеров может занять ресурсы, необходимые вашему сайту. А серверам с низкими вычислительными характеристиками может быть сложно справиться с объемом запросов, получаемых веб-сайтом. Все эти аспекты выходят далеко за рамки данной книги, но ошибочно полагать, что, если вы оптимизируете использование HTTP посредством применения некоторых из приведенных здесь советов и приемов, у вас никогда не возникнет проблем с производительностью.

Технологии повышения производительности, связанные с HTTP, важны, и любые ресурсы о веб-производительности (книги, блоги или лекции) в значительной степени концентрируют на них свое внимание. Однако данные технологии не являются конечной целью веб-производительности. Поэтому оптимизируйте использование HTTP разумно, ведь вы можете получить больше преимуществ посредством оптимизации других областей веб-сайта или веб-приложения.

6.4 Оптимизация и для HTTP/1.1, и для HTTP/2

Учитывая неплохую поддержку браузерами[1], HTTP/2 должен быть доступен большинству пользователей. Однако некоторые пользователи не используют HTTP/2, так как пользуются старыми браузерами или старыми устройствами, на которых обновление браузеров уже не поддерживается (например, старые мобильные телефоны). Также понижать уровень соединения между браузером и сервером могут прокси (включая антивирусное программное обеспечение). Что же делать этим пользователям, если на своем сайте вы используете какой-либо из методов, относящихся к HTTP/2, описанных в этой главе (например, минимизация конкатенации)? В лучшем случае веб-сайт будет доступен для таких пользователей, даже если вы начнете оптимизацию HTTP/2. В худшем случае при отказе от оптимизации HTTP/1 сайт будет работать медленнее. Однако так быть не должно. На самом деле вы можете оптимизировать свой веб-сайт как для HTTP/1, так и для HTTP/2.

6.4.1 Измерение трафика HTTP/2

Первое, что вам необходимо сделать, – это измерить объем трафика, который использует каждый протокол. Я предполагаю, что на этом этапе

[1] https://caniuse.com/#feat=http2.

вы уже перешли на HTTP/2, но еще не до конца поменяли свой сайт, поэтому оптимизация HTTP/1.1 все еще присутствует. Если подавляющая часть вашего трафика уже направлена на HTTP/2, нет смысла беспокоиться о трафике HTTP/1. Веб-сайт по-прежнему будет работать, однако уже немного медленнее, чем нужно.

Самый простой способ измерить трафик HTTP/2 – зарегистрировать его в журналах вашего веб-сервера. В Apache вы можете добавить эти данные в директиву `LogFormat`, обычно хранящуюся в основном файле httpd.conf (или в некоторых дистрибутивах в файле apache2.conf). Для этого в `LogFormat` необходимо добавить параметр `%H`:

```
LogFormat "%h %l %u %t %{ms}T %H \"%r\" %>s %b \"%{Referer}i\"
"%{User-Agent}i\" %{SSL_PROTOCOL}x %{SSL_CIPHER}x
%{Content-Encoding}o %{H2_PUSHED}e" combined
CustomLog /usr/local/apache2/log/ssl_access_log combined
```

В журналах доступа вы увидите следующее:

```
78.17.12.1234 - - [11/Mar/2018:22:04:47 +0000] 3 HTTP/2.0 "GET / HTTP/2.0"
200 1847 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_3)
    AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.186
    Safari/537.36" TLSv1.2 ECDHE-RSA-AES128-GCM-SHA256 br
```

Таким образом, мы видим, что протокол (`HTTP/2.0`) регистрируется еще до запроса (`GET / HTTP/2.0`). Поскольку Apache печатает в журналах запрос в формате HTTP/1, протокол можно получить в строке запроса (`%r`), но, вероятно, проще указать его отдельно в файле журнала, используя `%H`, таким образом вам будет легче проводить анализ.

У nginx регистрация протокола происходит с помощью переменной среды `$server_protocol`:

```
log_format my_log_format '$remote_addr - $remote_user [$time_local] '
                '$server_protocol "$request" $status $body_bytes_sent '
                '"$http_referer" "$http_user_agent"';
access_log /usr/local/nginx/nginx-access.log my_log_format;
```

Если вы пользуетесь каким-либо другим веб-сервером, найдите информацию об этом в его документации.

Каково значение пограничного сервера?

Если вы используете балансировщик нагрузки перед несколькими веб-серверами, вам может потребоваться измерить трафик протокола в самом балансировщике. Процесс измерения зависит от типа вашего балансировщика нагрузки.

Балансировщик нагрузки HTTP (также называемый балансировщиком нагрузки уровня 7, следуя модели OSI, упомянутой в главе 1) завершает HTTP-соединение и устанавливает другое HTTP-соединение от балансировщика нагрузки к веб-серверу. Следовательно, если вы измеряете этот протокол на веб-сервере, его журналы показывают протокол, используемый для соединения балансировщика нагрузки с веб-сервером, однако он может не совпа-

> дать с основным соединением (клиент – балансировщик нагрузки), которое вас, вероятно, интересует больше всего. В этом случае вам следует измерять трафик протокола на балансировщике нагрузки, а не на веб-сервере.
>
> Балансировщик нагрузки TCP (также известный как балансировщик нагрузки уровня 4) работает на уровне TCP и пересылает полезную нагрузку пакетов TCP (HTTP-сообщения) на следующие веб-серверы. Следовательно, HTTP-сообщения являются исходными сообщениями, и протокол можно измерить на уровне веб-сервера.
>
> Так называемый пограничный сервер является точкой входа для пользователя интересующего вас протокола (в данном случае HTTP), поэтому, когда вы пытаетесь измерить свой трафик, важно хорошо знать инфраструктуру и выбрать соответствующее место для измерения.

При журналировании использования протоколов вы можете проанализировать журналы и увидеть процент трафика каждого из протоколов. Если вы работаете в системе на базе Linux или UNIX, вы можете воспользоваться комбинациями `grep`, `sort` и `uniq`:

```
$ grep -oh 'HTTP\/[0-9]\.[0-9]*' ssl_access_log | sort | uniq -c
    196 HTTP/1.0
   1182 HTTP/1.1
   5977 HTTP/2.0
```

Здесь мы видим наличие трафика HTTP/1.0 (который зачастую создается ботами). Трафик HTTP/1.1 составляет 16 %, а HTTP/2 – 81 %.

6.4.2 Отслеживание поддержки HTTP/2 на стороне сервера

Если вы предполагаете, что пользователи, использующие HTTP/1, по-прежнему составляют значительную часть вашего веб-трафика, вы можете определить, выполняется ли текущее соединение через HTTP/1.1 или HTTP/2, и создать для каждого типа разные ответы. Пользователи HTTP/1 могут иметь полностью объединенные ресурсы и даже загружать ресурсы из сегментированных доменов, тогда как пользователи HTTP/2 могут получать меньше объединенных ресурсов и загружать все из основного домена.

Чтобы обрабатывать два протокола по-разному, необходимо знать, какой протокол использует входящее соединение. Как и при измерении использования протокола, вам необходимо измерить трафик протокола на вашем пограничном сервере (см. сноску выше, «Каково значение пограничного сервера?»), что зависит непосредственно от его возможностей. Возможно, данную информацию необходимо будет передать далее по соединению.

Большинство веб-серверов устанавливает переменные среды, которые можно использовать для принятия решений и изменения конфигурации. Сценарии CGI и PHP могут обращаться к переменной среды `SERVER_PROTOCOL`, для которой необходимо установить значение HTTP/1.1 или HTTP/2.0 в зависимости от ситуации.

Некоторые веб-серверы устанавливают дополнительные переменные. В табл. 6.7 приведены переменные среды[1], которые можно использовать на сервере Apache.

Таблица 6.7 Переменные среды Apache HTTP/2

Имя переменной	Тип значения	Описание
HTTP2	Флажок	HTTP/2 уже используется
H2PUSH	Флажок	Для этого соединения включен push-сервер HTTP/2, который также поддерживается клиентом
H2_PUSH	Флажок	Альтернативное название H2PUSH
H2_PUSHED	Строка	Переменная пуста или имеет значение PUSHED для запроса, отправляемого сервером
H2_PUSHED_ON	Число	Определяет номер потока HTTP/2, который инициировал отправку запроса
H2_STREAM_ID	Число	Номер потока HTTP/2 запроса
H2_STREAM_TAG	Строка	Представляет собой уникальный идентификатор потока процесса HTTP/2, состоящий из идентификатора соединения и идентификатора потока, разделенных дефисом

Данные переменные среды доступны для конфигурации Apache, а также для сценариев CGI и PHP. Их также можно использовать в директивах `LogFormat`, но тогда нужно добавить часть `%{H2_ PUSHED}e` в пользовательский формат журнала, как я сделал ранее, для отслеживания push- ресурсов в файлах журнала.

Веб-сервер nginx имеет переменную `$http22`, для которой установлено значение `h2`, если HTTP/2.0 используется через HTTPS (как и все соединения браузера), или `h2c`, когда используются незашифрованные соединения HTTP/2. На момент написания этой книги nginx не предоставляет дополнительных переменных среды, например функцию push.

Многие также используют Apache и nginx в качестве обратных прокси-серверов и для осуществления прокси-запросов к нижестоящему приложению (например, Node или серверу приложений на основе Java, например Tomcat). В Apache используют директиву `ProxyPass`:

```
ProxyPass /webapplication/ http://localhost:3000/
```

В этом сценарии из-за этого у систем ниже по потоку не будет доступа к переменным среды Apache. Однако можно установить дополнительные HTTP-заголовки для информирования нижестоящей системы с помощью директивы `RequestHeader`:

```
#Определение переменной HTTP_VERSION, так как в Apache ее нет
#(SERVER_PROTOCOL и Request_Protocol не являются переменными полного окружения)
SetEnvIf Request_Protocol "(.*)" HTTP_VERSION=$1
#Затем используем эту переменную для установки заголовка HTTP для нижестоящих систем
RequestHeader set protocol "%{HTTP_VERSION}e"
#Добавляем некоторые преднастроенные переменные HTTP2
```

1 https://httpd.apache.org/docs/2.4/mod/mod_http2.html#envvars.
2 http://nginx.org/en/docs/http/ngx_http_v2_module.html#variables.

```
RequestHeader set http2 %{HTTP2}e
RequestHeader set h2push %{H2PUSH}e
ProxyPass /webapplication/ http://localhost:3000/
```

Приложения ниже по потоку могут читать эти HTTP-заголовки, как и любые другие.

Аналогичный синтаксис для nginx:

```
location /webapplication/ {
    proxy_set_header protocol $server_protocol;
    proxy_set_header http2 $http2;
    proxy_pass http://localhost:3000;
}
```

Используя сервер Node в качестве примера, вы можете вернуться к простому серверу из главы 5 и добавить две дополнительные строки для регистрации поддержки HTTP/2, как показано в листинге 6.1.

Листинг 6.1 Сетевой узел с проверкой HTTP-заголовка

```
'use strict'

var http = require('http')
const port = 3000

const requestHandler = (request, response) => {
  const { headers } = request;
  console.log('HTTP Version: ' + headers['protocol'])
  console.log('HTTP2 Support: ' + headers['http2'])
  console.log('HTTP2 Push Support: ' + headers['h2push'])
  response.setHeader('Link','</assets/css/common.css>;rel=preload')
  response.writeHead(200, {"Content-Type": "text/html"})
  response.write('<!DOCTYPE html>\n')
  response.write('<html>\n')
  response.write('<head>\n')
  response.write('<link rel="stylesheet" type="text/css"
href="/assets/css/common.css">\n')
  response.write('</head>\n')
  response.write('<body>\n')
  response.write('<h1>Test</h1>\n')
  response.write('</body>\n')
  response.write('</html>\n')
  response.end();
}

var server = http.createServer(requestHandler)
server.listen(port)
console.log('Server is listening on ' + port)
```

При посещении `/webapplication/` из браузера через правильно настроенный сервер Apache в журналы узла выводится следующее:

```
HTTP Version: HTTP/2.0
HTTP2 Support: on
HTTP2 Push Support:on
```

Браузеры HTTP/1 выводят следующее:

```
HTTP Version: HTTP/1.1
HTTP2 Support: (null)
HTTP2 Push Support: (null)
```

В nginx мы видим несколько другое:

```
HTTP Version: HTTP/2.0
HTTP2 Support: h2
HTTP2 Push Support: undefined
```

Запрос HTTP/1.1 через nginx выглядит так:

```
HTTP Version:HTTP/1.1
HTTP2 Support:undefined
HTTP2 Push Support:undefined
```

Альтернативным методом является передача деталей в виде параметров запроса, а не HTTP-заголовков. Однако я считаю, что HTTP-заголовки понятнее и их легче добавлять как для запросов `GET`, так и для `POST`. Тем не менее вы видите, что можно узнать версию протокола, и в зависимости от нее ваше приложение будет реагировать по-разному.

6.4.3 Отслеживание поддержки HTTP/2 на стороне клиента

Клиентским приложениям также может понадобиться информация об используемом вами протоколе (HTTP/1 или HTTP/2). На сегодняшний день стандартизированного способа получить ее не существует, однако API-интерфейс Resource Timing Level 2 включает атрибут `nextHopProtocol`, который может вам помочь решить эту задачу.

Однако понять, что именно клиент считает доступным, сложно из-за наличия промежуточных прокси. Возможно, веб-браузер ограничен HTTP/1.1, но прокси-сервер подключается через HTTP/2 к серверу (хотя, скорее всего, происходит наоборот, браузер поддерживает HTTP/2, но ограничен HTTP/1.1 из-за прокси). По этой причине я считаю, что сначала лучше понять, какой протокол используется на стороне сервера, а затем отправить результаты обратно клиенту. Вы можете отправить результаты несколькими способами, например посредством HTTP-заголовка или переменной JavaScript. Единственное предостережение – вам следует учитывать, как кеширование ресурса, указывающего эту информацию, может повлиять на последующие соединения. Если вы запускаете соединение HTTP/2 и кешируете ресурс, который клиентская сторона использует для обозначения соединения, а затем клиент переключается на соединение HTTP/1, возможно, вы выполняете оптимизацию неправильно.

[1] https://www.w3.org/TR/resource-timing-2/#dom-performanceresourcetiming-nexthopprotocol.

6.4.4 Объединение соединений

Согласно спецификации HTTP/2 одно и то же соединение можно использовать для нескольких доменов, если они являются *доверенными*[1], т. е. если домены разрешаются на один и тот же IP-адрес, а сертификат HTTPS охватывает оба домена. Делается это для того, чтобы максимизировать единое соединение и разрешить автоматическое объединение доменов, размещенных на одном сервере (также известное как *объединение соединений*).

Предположим, что сайт www.example.com использует для размещения изображений домен images.example.com. Если оба этих домена размещены на одном сервере и представляют собой отдельные виртуальные хосты, в HTTP/2 они могут обслуживаться одним и тем же соединением при условии установки соответствующего *псевдозаголовка* `:authority`. Такой сценарий обычно происходит, если *сегментированные* домены создаются исключительно по причине неэффективности HTTP/1 и для них не используется отдельный сервер, как показано на рис. 6.15.

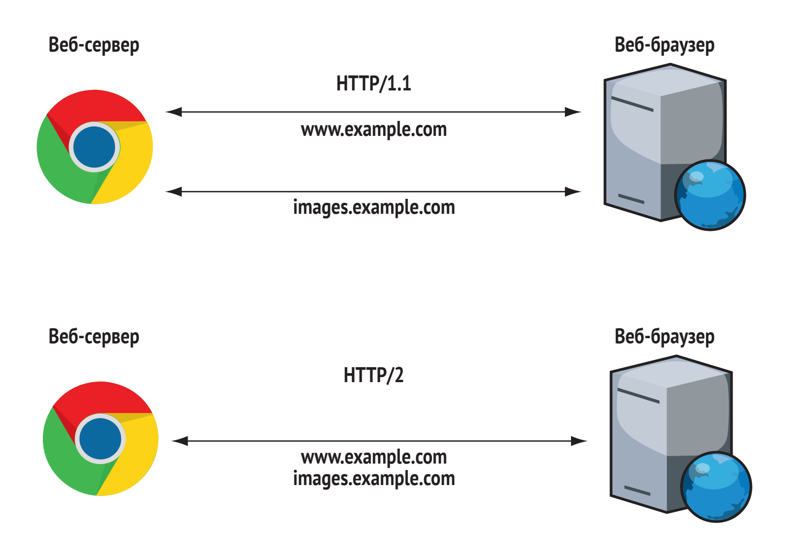


Рис. 6.15 Объединение соединений по HTTP/2

На более высоком уровне (например, в инструментах разработчика браузера) обычные HTTP-запросы выглядят точно так же, как и сегментированные; клиент может решить объединить их только на уровне соединения. У многих веб-сайтов могут по-прежнему существовать сегментированные домены, в которых HTTP/1.1-соединения будут использоваться автоматически, а HTTP/2-соединения будут объединяться автоматически, действуя так, как если бы они были не доверенными доменами, и все это будет обслуживаться через одно соединение. Казалось бы, такая ситуация – это идеальный сценарий, который не требу-

[1] https://httpwg.org/specs/rfc7540.html#reuse.

ет дополнительных усилий для продолжения поддержки пользователей HTTP/1 или оптимизации для пользователей HTTP/2. Однако в реальности, как всегда, все немного сложнее.

Начнем с того, что данный сценарий становится возможным, только если все домены расположены на одном сервере. Если домены расположены на разных серверах, их должны обслуживать отдельные соединения. Также в браузере должно быть реализовано повторное использование соединения (что есть сейчас не во всех браузерах)[1]. В спецификации говорится, что соединение *может быть* использовано повторно, но это *необязательно*. На момент написания этой книги Safari и Edge, в отличие от Chrome и Firefox, не поддерживают данную функцию.

Кроме того, при определенных реализациях, объединение соединений может вызвать ряд проблем. Ввиду наличия нескольких IP-адресов для доменов (некоторые из которых могут использоваться совместно с другими доменами) браузер может решить использовать соединение повторно (или объединить несколько соединений), когда на самом деле это не представляется возможным. Рассмотрим IP-адреса, указанные в табл. 6.8.

Таблица 6.8 Пример объединения соединений

Домен	IP-адреса
www.example.com	1.2.3.4 1.2.3.5
images.example.com	1.2.3.4 1.2.3.6

В этом случае по желанию клиента любое HTTP/2-соединение по IP-адресу 1.2.3.4 может обслуживать запросы как www.example.com, так и images.example.com, но соединение по другим IP-адресам (1.2.3.5 и 1.2.3.6) они осуществить не могут. Firefox реализовал так называемое агрессивное объединение соединений. Согласно этому методу браузер, если он будет информирован об объединении, будет пытаться использовать любое соединение для обоих доменов независимо от того, какой из трех IP-адресов используется для соединения. Подобные ситуации приводят к ошибкам, как при переходе на HTTP/2 заметила компания BBC[2].

Код состояния `421` был создан, чтобы позволить серверу вежливо сообщать браузеру, что он использовал неправильное соединение, и заставить его обратить внимание на то, куда он должен отправлять эти запросы. Однако, как выяснила BBC, поддержка этого кода все еще ограничена. В качестве альтернативного решения существует фрейм `ORIGIN`[3], который позволяет серверу информировать клиента о том, для каких доменов он является *доверенным*, а не действовать наугад. На момент написания этой

[1] https://daniel.haxx.se/blog/2016/08/18/http2-connection-coalescing/.

[2] https://medium.com/bbc-design-engineering/http-2-is-easy-just-turn-it-on-34baad2d1fb1.

[3] https://tools.ietf.org/html/rfc8336.

книги данный фрейм – новинка, однако его уже поддерживают некоторые серверы[1,2], а другие имеют открытые запросы на его отслеживание[3,4]. Со стороны браузеров фрейм поддерживает Firefox[5], и ожидается, что за ним последуют другие браузеры и впоследствии фрейм будет включен в рекомендуемый стандарт. Как правило, `ORIGIN` отправляется в начале соединения (в идеале после `SETTINGS`), что также должно предотвратить отправку неверных запросов. В связи с этим проверка подлинности вторичного сертификата в HTTP/2 – это еще одно предложение, которое позволит объединить соединения, даже если на одном IP-адресе используются разные сертификаты.

Иными словами, объединение соединений – это очень сложно, поэтому полагаться на этот метод я не советую. Вместо этого обратите внимание на то, требуется ли вашим доменам сегментирование и позволит ли это получить прирост производительности. Если сегментирование вам все же требуется, возможно, лучше разместить такие домены на отдельных серверах, чтобы предотвратить проблемы с объединением.

6.4.5 Сколько времени занимает оптимизация для пользователей HTTP/1.1

Еще одна вещь, которую нужно учитывать при оптимизации для всех групп пользователей, – это время. Оптимизация требует дополнительных усилий, поэтому подумайте, готовы ли вы выполнять дополнительную работу, и если да, то сколько времени вы готовы этому уделить. Конечно, существуют и альтернативы, такие как отказ от отмены обходных путей HTTP/1.1, поскольку при HTTP/2 в них теряется необходимость, однако работают они не менее эффективно. Также, если большая часть ваших посетителей уже использует HTTP/2, лучше оптимизировать для большинства за счет меньшинства и позволить тем, кто использует соединения HTTP/1.1, загружать страницы медленнее, чем необходимо. Каждый владелец веб-сайта должен принять взвешенное решение, исходя из своей базы и ожидаемых последствий изменений. Поскольку браузер поддерживает протокол HTTP/2, пользователи, которые не могут использовать HTTP/2, делятся на несколько категорий:

- пользователи с устаревшими версиями программного обеспечения (в зависимости от того, насколько вы поддерживаете старые версии, в любом случае некоторые функции для таких пользователей будут недоступны);
- пользователи, выходящие в Internet с помощью корпоративных прокси (вероятно, при более быстром подключении);

1 https://github.com/nghttp2/nghttp2/pull/901.
2 https://github.com/h2o/h2o/pull/1199.
3 https://github.com/icing/mod_h2/issues/96.
4 https://trac.nginx.org/nginx/ticket/1530.
5 https://bugzilla.mozilla.org/show_bug.cgi?id=1337791.

- пользователи, применяющие антивирусные прокси (скорее всего, это пользователи стационарных компьютеров, в основном пользующиеся широкополосным доступом);
- те, кто пользуются непопулярными браузерами (у которых могут быть другие проблемы с отображением вашего сайта);
- боты и программы автоматического сбора информации (которые, вероятно, наименее важны для вас).

В конце концов, стоит оценить количество посетителей вашего сайта, все еще использующих HTTP/1, а затем решить, стоит ли проводить меры по оптимизации для обоих протоколов. Для крупных сайтов ответ, вероятно, будет положительным, но для небольших оптимизация может быть нецелесообразной. Даже если вы не произведете оптимизацию, сайт по-прежнему будет работать (хотя и медленнее). Подобным образом постепенное сокращение возможностей используется в дизайне веб-сайтов с целью создать работающий, но неоптимизированный сайт для браузеров, которые не поддерживают функции, необходимые для оптимального взаимодействия с ним.

Резюме

- HTTP/2 был разработан для устранения неэффективности HTTP/1.1.
- Многие надеялись, что оптимизация производительности HTTP/1.1, которая требует усилий и имеет свои недостатки, больше не потребуется, но их ожидания сбылись лишь частично. Сейчас необходимость в таких методах меньше, однако отказываться от них пока что не стоит.
- Другие методы повышения веб-производительности в основном остаются актуальными для HTTP/2, но, возможно, стоит пересмотреть их для вашего сайта при переходе на HTTP/2.
- Возможна оптимизация как для соединений HTTP/1.1, так и для HTTP/2.
- Объединение соединений позволяет браузеру автоматически избавляться от сегментированных доменов, однако этот процесс очень сложен.

Часть III

Продвинутый уровень использования HTTP/2

В первой части этой книги мы познакомились с протоколом HTTP/2. Во второй части представлены детали работы протокола и примеры его использования на практике. Основы HTTP/2 и принцип его работы описаны в главе 4; глава 5 посвящена абсолютно новой для протокола концепции HTTP/2 push; а в главе 6 рассказывается о реализации HTTP/2 на практике, а также о его влиянии на процесс разработки.

В третьей части я копну несколько глубже и расскажу о некоторых сложных аспектах протокола, доступных лишь продвинутым пользователям. В главе 7 мы рассмотрим ранее упущенные части спецификации, а глава 8 в отдельном порядке расскажет нам о спецификации сжатия HTTP-заголовка HPACK. Вышеупомянутые главы помогут вам перейти с уровня «прочной основы» на уровень экспертного понимания протокола HTTP/2. Благодаря этому вы сможете решить любую соответствующую проблему на своем сайте и, возможно, даже внести свой вклад в дальнейшее развитие протокола!

7

Расширенные возможности HTTP/2

В этой главе мы рассмотрим:

- концепцию состояний HTTP/2-потока;
- управление потоками информации;
- приоритизацию в HTTP/2;
- проверку на совместимость с HTTP/2.

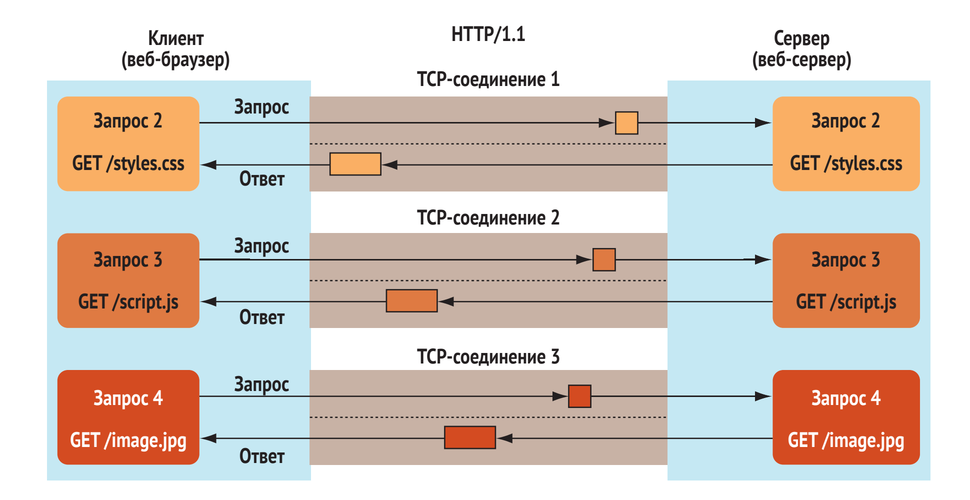
Данная глава посвящена упущенным мною ранее частям протокола HTTP/2. Я расскажу о них в порядке, примерно соответствующему их описанию в спецификации[1]. Многие аспекты, рассматриваемые здесь, сложны постольку, поскольку зачастую они не контролируются напрямую ни веб-разработчиками, ни администраторами отдельных серверов (за исключением случаев, когда те сами пишут HTTP/2-сервер с нуля). Однако понимание этих аспектов обеспечит вам и более глубокое понимание самого протокола, а если вы захотите реализовать свой собственный HTTP/2-сервер, эта информация может помочь вам при его отладке. Кроме того, возможно, что в будущем разработчики или, по крайней мере, администраторы веб-серверов все же получат контроль над этими частями протокола. В главе 8 я расскажу вам о протоколе HPACK, для которого существует отдельная спецификация.

---

[1] https://httpwg.org/specs/rfc7540.html.

7.1 Состояния HTTP/2-потока

Отдельный HTTP/2-поток создается при каждой загрузке, а по ее завершении отбрасывается. В этом состоит основное различие соединений HTTP/1 и потоков HTTP/2, ввиду чего вторые не могут быть аналогом первых. Я привожу именно такой пример с целью наиболее доступно представить данные концепции и разницу между ними. На многих диаграммах (например, как в главе 2 этой книги, которые я продублирую на рис. 7.1) между потоками HTTP/2 и соединениями HTTP/1 проведены параллели, однако это не совсем верно, поскольку потоки, в отличие от соединений, не используются повторно.



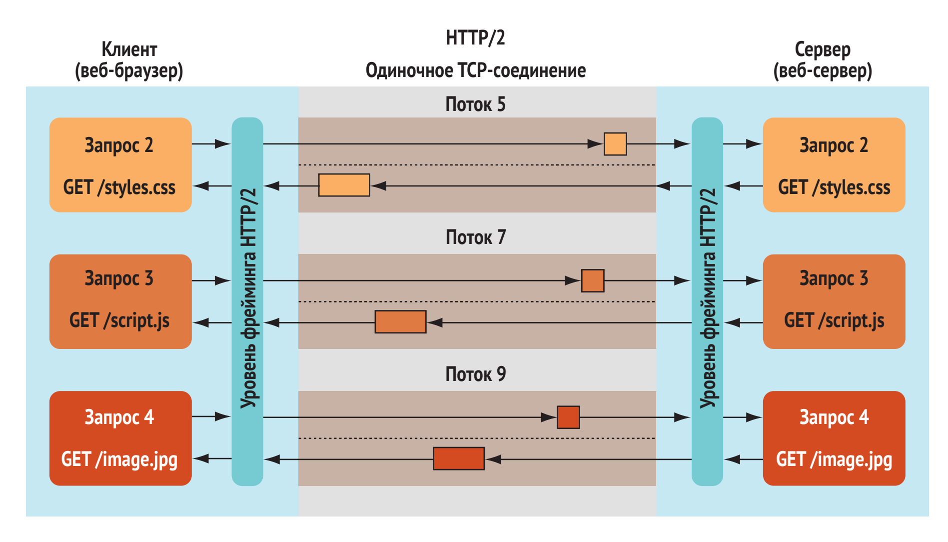


Рис. 7.1 Соединения HTTP/1.1 и потоки HTTP/2 могут быть похожи, однако на самом деле они абсолютно разные

По завершении доставки ресурса потоки закрываются, а при поступлении запроса на новый ресурс запускается новый поток. Потоки – это виртуальная концепция, представляющая собой не что иное, как номер, которым помечен каждый фрейм; такие номера называются идентификаторами потока. Закрытие и открытие потока – процесс гораздо менее ресурсоемкий, нежели закрытие и открытие соединения HTTP/1 (которое включает в себя трехстороннее рукопожатие TCP и иногда согласование протокола HTTPS перед отправкой запроса). HTTP/2-соединения намного сложнее соединений HTTP/1, поскольку перед осуществлением запроса они требуют «волшебной» преамбулы HTTP/2 и хотя бы одного фрейма `SETTINGS`. Таким образом, намного проще использовать именно HTTP/2-потоки.

Потоки HTTP/2 имеют свой жизненный цикл, на протяжении которого они принимают различные состояния. Фрейм `HEADERS`, отправленный от клиента, начинается как HTTP-запрос (такой, как `GET`) и проходит несколько состояний:

- *неактивный поток*. В неактивном состоянии поток находится сразу после его создания или же когда на него ссылаются. На самом деле большинство потоков не остается в этом состоянии надолго. Объясняется это тем, что мало кто ссылается на потоки, если в будущем не собирается их использовать. Именно поэтому незанятые потоки немедленно входят в оборот и становятся активными, после чего сразу же переходят к следующей фазе: открытию;
- *открытый поток*. Поток считается открытым и становится доступным обеим сторонам после отправки запроса фрейма `HEADERS`. Он остается в этой фазе, пока клиент не отправит все необходимые данные. Поскольку в HTTP/2 в одном фрейме `HEADERS` может быть отправлено большинство запросов, почти всегда после этого этапа поток сразу же переходит в следующую фазу (полузакрытый поток);
- *полузакрытый поток*. Поток становится полузакрытым, когда с помощью флажка `END_STREAM` клиент указывает, что фрейм `HEADERS` содержит все необходимые данные. После этого поток может быть использован только в одностороннем порядке: для отправки ответов клиенту; таким образом, его не следует использовать для отправки данных от клиента (за исключением управляющих фреймов, таких как `WINDOW_UPDATE`);
- *закрытый поток*. После отправки сервером последнего фрейма с флажком `END_STREAM` поток считается закрытым и становится непригодным для дальнейшего использования.

Несмотря на то что в списке выше приведены переходы между состояниями обычного HTTP-запроса, инициированного клиентом, тот же путь может проходить и запрос, инициированный сервером. На сегодняшний день в число таких запросов входит только HTTP/2 push, однако в будущем их список может пополниться. Для запросов, инициированных сервером, запускаются отдельные потоки (обещанный идентификатор потока), которые проходят аналогичный цикл состояний:

- *неактивное.* В этом состоянии обещанный поток находится сразу после его создания или на него ссылается фрейм PUSH_ PROMISE, отправленный в другом потоке;
- *зарезервированное.* Далее поток переходит в состояние зарезервированного, в котором он будет находиться до тех пор, пока сервер не будет готов отправить ресурс. Таким образом, вы знаете о существовании этого потока (и о том, что он простаивает) и о его назначении (поэтому скорее не простаивает, а является зарезервированным для определенного ресурса, откуда и пришло название), но при этом не имеете полного представления о ресурсе, для которого этот поток предназначен. Такую ситуацию мы можем наблюдать после получения фрейма HEADERS в первом примере. Поток не может находиться в открытом состоянии, поскольку он предназначен для push-ресурса и клиент априори не может отправить данные по этому потоку. Таким образом, он находится в зарезервированном состоянии, а затем при отправке фрейма HEADERS (после отправки фрейма PUSH_PROMISE в исходном потоке) переходит в полузакрытое состояние; и такой порядок фаз неслучаен;
- *полузакрытое.* Поток переходит в полузакрытое состояние, когда сервер начинает push-загрузку ответа. В это время он может быть использован только для отправки соответствующего push-ресурса;
- *закрытое.* Поток считается закрытым по завершении отправки ресурса сервером, когда в последнем фрейме DATA он использует флажок END_STREAM. После этого поток больше не должен быть использован.

На рис. 7.2 представлена схема всех возможных состояний потока HTTP/2. В ней приведены два вышеупомянутых цикла, а также другие сценарии (например, когда фрейм RST_STREAM используется для преждевременного закрытия соединения).

В каждом из этих потоков информации у клиента и сервера разнится представление о состоянии потока. Оно зависит от того, какая сторона инициирует запрос, а какая переходит в это состояние на основании сообщения от другой. Именно поэтому у некоторых из этих состояний есть локальный или удаленный индикатор (в зависимости от того, являетесь ли вы инициатором или получателем потока соответственно). Кроме того, для каждого состояния существуют переходы send и recv.

Итак, мы уже знаем, что запрос GET проходит через следующие состояния: неактивное, открытое, полузакрытое и закрытое. Уделим особое внимание неоднозначному полузакрытому состоянию: для клиента оно закрыто (ввиду чего он может только получать данные), а для сервера полузакрыто. Клиент видит поток как наполовину закрытый (локальный), а сервер видит его как наполовину закрытый (удаленный). Следовательно, запрос для клиента и сервера проходит через разные циклы состояний; если вы посмотрите на него со стороны сервера, он будет проходить, согласно правой части схемы на представленном рисунке, если со стороны клиента – по левой.

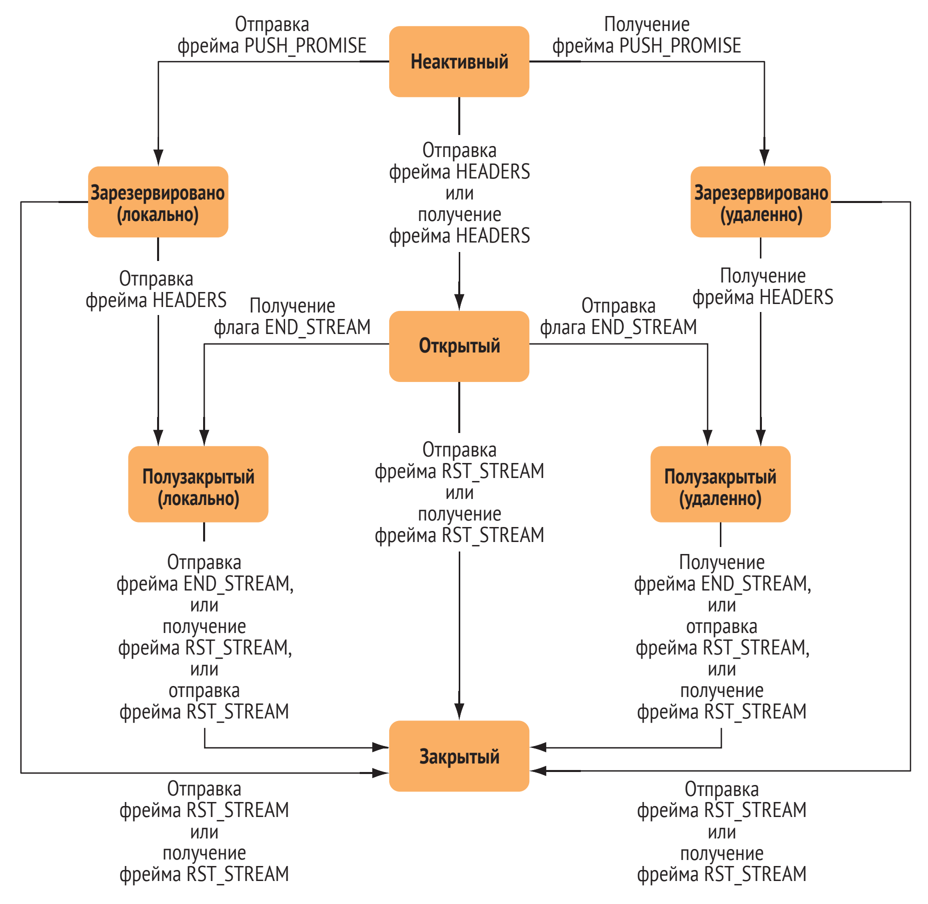


Рис. 7.2 Схема состояний потока HTTP/2

Также стоит отметить, что схема состояний отражает лишь переходы между состояниями. Однако некоторые фреймы не влекут за собой его смену. Например, фреймы `CONTINUATION` считаются расширениями предыдущих фреймов `HEADERS`, поэтому на схеме они входят в состав `HEADERS`. Точно так же существуют фреймы (такие как `PRIORITY`, `SETTINGS`, `PING` и `WINDOW_UPDATE`), которые вообще не приводят к смене состояния, и поэтому они не отражены в схеме.

Откровенно говоря, данная схема состояний важна не столько для пользователей, сколько для разработчиков HTTP/2-библиотек низкого уровня. С ее помощью последние могут понять, какие фреймы необходимо отправлять при определенной фазе состояния. Схема составлена согласно спецификации HTTP/2[1], где можно найти множество ссылок на информацию о различных состояниях. Любые попытки смены состояния, отклоняющиеся от спецификации, приводят к появлению ответа `PROTOCOL_ERROR`. Если вы хорошо разберетесь в схеме состояний, то без

[1] https://httpwg.org/specs/rfc7540.html#StreamStates.

труда сможете найти причину подобных ошибок (однако такие ошибки присущи базовым реализациям HTTP/2, и исправить их самостоятельно зачастую не представляется возможным).

Поначалу схема может казаться пугающей, особенно в сравнении с концепциями, о которых я рассказывал ранее. Ничего подобного вы не найдете в инструментах разработчика браузера или в каких-либо других инструментах, описанных в этой книге (например, nghttp и Wireshark). Состояния потока – это внутренний аспект протокола, а от реализаций HTTP/2 требуется только его отслеживание и поддержка. Учитывая данный факт, ситуация становится еще сложнее. Как правило, все упрощает один простой шаг – возврат к основному варианту использования (запрос ресурса HTTP); этот процесс я описывал ранее.

7.2 Управление потоками информации

Управление потоками информации является важным аспектом сетевых протоколов. С его помощью получатель может запретить отправку ему ресурсов, если он еще не готов к их обработке (например, если он слишком загружен для обработки дополнительных данных). Ввиду того, что клиенты обрабатывают данные с разной скоростью, управление потоками – чрезвычайно полезный механизм. Высокоскоростные серверы отправляют данные очень быстро, но если они работают с клиентом, чья скорость значительно меньше (например, мобильное устройство), то последний вряд ли справится со столь объемным потоком данных. Из-за этого клиент начнет буферизировать данные в памяти, а когда буфер заполнится, он будет отбрасывать пакеты и впоследствии запрашивать их повторную отправку. В результате такой сценарий становится слишком ресурсоемким и для серверной, и для клиентской стороны, а также для самой сети.

В HTTP/1.1 управление потоками не требовалось, поскольку в соединении всегда находилось только одно сообщение. Таким образом, на уровне соединения можно было использовать *управление передачей TCP*. Если получатель прекращает прием TCP-пакетов, то вторая сторона прекращает отправлять их ввиду того, что его окно загрузки (CWND) увеличивается до максимально возможного размера (см. главу 2).

HTTP/2 позволяет нам объединять независимые потоки в одно *мультиплексированное соединение*, поэтому управления потоком информации только на уровне соединения становится недостаточно. Оно должно осуществляться в том числе и на уровне потока, поскольку для вас, например, может быть важнее получить больше данных именно от какого-либо определенном потока. В главе 4 приведен пример веб-сайта с размещенным на нем видео, воспроизведение которого было приостановлено пользователем. В подобном случае сайту может быть необходимо остановить дальнейшую загрузку медиафайла на одном потоке, но при этом разрешить функционирование других потоков в HTTP/2-соединении.

Управление потоком информации в HTTP/2 схоже с работой TCP. В начале соединения (с помощью фрейма SETTINGS) определяется размер окна управления потоком (если размер не указан, используется значение по умолчанию 65 535 октетов). Затем каждая отправленная часть данных вычитается из этой суммы, после чего обратно добавляется каждый бит подтвержденных данных с помощью фрейма WINDOW_UPDATE. На уровне соединения существует окно управления потоком, которое как бы дублирует окно управления потоком TCP. Также такие окна существуют и для каждого потока в отдельности. Размер отправляемого ресурса не должен превышать максимальный размер наименьшего окна управления потоком (на одном из уровней). Когда размер этого окна достигает нуля, отправителю следует прекратить передачу данных до тех пор, пока он не получит соответствующее подтверждение. Если вы реализуете клиент или сервер на HTTP/2 и забудете реализовать фреймы WINDOW_UPDATE, совсем скоро вы заметите, что третья сторона прекратит «общаться» с вами.

Управление потоком необходимо преимущественно для фреймов DATA (хотя другие типы фреймов HTTP/2 также могут попадать под в зону его влияния). Фреймы управления (и в частности WINDOW_UPDATE, необходимые для управления потоком) могут быть отправлены даже тогда, когда клиент предупреждает о превышении максимального размера окна управления.

7.2.1 Пример управления потоками информации

В качестве примера управления потоком вернемся к инструменту nghttp. В этом разделе мы инициируем запрос домашней страницы Facebook и всех необходимых ей ресурсов, а затем с помощью утилиты grep мы выделим и просмотрим только нужные нам строки кода.

```
$ nghttp -anv https://www.facebook.com | grep -E "frame
<|SETTINGS|window_size_increment"
[  0.110] recv SETTINGS frame <length=30, flags=0x00, stream_id=0>
          [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
[  0.110] recv WINDOW_UPDATE frame (window_size_increment=10420225)
          (window_size_increment=10420225)
[  0.110] send SETTINGS frame <length=12, flags=0x00, stream_id=0>
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
```

Здесь мы можем видеть, что сервер Facebook принял размер окна управления потоком 65 536 октетов (значение SETTINGS_INITAL_WINDOW_SIZE во фрейме recv SETTINGS), а nghttp определил значение 65 535 октетов (значение SETTINGS_INITAL_WINDOW_SIZE во фрейме SETTINGS отправителя). К слову, 65 535 является размером по умолчанию, поэтому nghttp мог

его не отправлять. Таким образом, этот пример показывает, что стороны могут принимать разные размеры окна управления потоком (хотя здесь они достаточно близки и отличаются всего на один октет).

Между двумя фреймами SETTINGS находится первый фрейм WINDOW_UPDATE (выделен в коде):

```
[  0.110] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=10420225)
```

В этом кадре указано, что Facebook готов принять до 10 420 225 октетов данных, и, поскольку фрейм был отправлен в потоке 0, данный предел будет актуален для всего соединения, а также для всех потоков в дополнение к их собственному лимиту. В потоке 0 не должно быть фреймов DATA, вследствие чего он не требует управления. Таким образом, его можно использовать для управления потоком на уровне соединения. Количество 10 420 225 больше, чем максимально возможный размер по умолчанию (65 535), и поэтому Facebook мог бы сложить эти значения, что повысило бы лимит до 10 485 761. Однако здесь приемлемы оба варианта.

Затем nghttp подтверждает настройки сервера, после чего следует еще несколько фреймов, в которых он устанавливает приоритеты (кстати, это один из немногочисленных случаев, когда фрейм может быть создан в неактивном состоянии и оставаться в нем до тех пор, пока не будет использован):

```
[  0.110] send SETTINGS frame <length=0, flags=0x01, stream_id=0>
[  0.110] send PRIORITY frame <length=5, flags=0x00, stream_id=3>
[  0.110] send PRIORITY frame <length=5, flags=0x00, stream_id=5>
[  0.110] send PRIORITY frame <length=5, flags=0x00, stream_id=7>
[  0.110] send PRIORITY frame <length=5, flags=0x00, stream_id=9>
[  0.110] send PRIORITY frame <length=5, flags=0x00, stream_id=11>
```

Что касается фреймов PRIORITY, сейчас их обсуждение можно упустить, поскольку я все равно расскажу о них позже.

Следующим шагом является отправка первого запроса в потоке 13 с помощью фрейма HEADERS:

```
[  0.110] send HEADERS frame <length=43, flags=0x25, stream_id=13>
```

Напомним, что потокам, инициированным клиентом, присваиваются идентификаторы потоков с нечетными номерами. Таким образом, ближайшим свободным потоком является поток 13, поскольку 11 использовался фреймом PRIORITY.

После этого сервер подтверждает фрейм SETTINGS и второй фрейм WINDOW_UPDATE, таким образом увеличивая размер окна потока 13 до 10 420 224 октетов (как ни странно, на один октет меньше, чем размер окна уровня соединения, но нам ведь и не сказано, что размеры должны быть одинаковыми):

```
[  0.134] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
[  0.134] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=13>
          (window_size_increment=10420224)
```

Затем nghttp получает фреймы ресурса HEADERS и DATA:

```
[  0.348] recv HEADERS frame <length=293, flags=0x04, stream_id=13>
[  0.349] recv DATA frame <length=1353, flags=0x00, stream_id=13>
[  0.350] recv DATA frame <length=2571, flags=0x00, stream_id=13>
[  0.351] recv DATA frame <length=8144, flags=0x00, stream_id=13>
[  0.374] recv DATA frame <length=5563, flags=0x00, stream_id=13>
[  0.375] recv DATA frame <length=2572, flags=0x00, stream_id=13>
[  0.376] recv DATA frame <length=1491, flags=0x00, stream_id=13>
[  0.377] recv DATA frame <length=2581, flags=0x00, stream_id=13>
[  0.378] recv DATA frame <length=4072, flags=0x00, stream_id=13>
[  0.379] recv DATA frame <length=5572, flags=0x00, stream_id=13>
```

После этого nghttp сообщает серверу количество полученных данных. Суммарный размер всех фреймов DATA составляет 33 919 октетов (1353 + 2571 + 8144 + 5563 + 2572 + 1491+ 2581 + 4072 + 5572). Именно это значение nghttp передает на сервер, причем оно присваивается и для уровня соединения (поток 0), и для потока 13:

```
[  0.379] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=33919)
[  0.379] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=13>
          (window_size_increment=33919)
```

Важно отметить, что при подсчете учитывается только длина информационного наполнения кадра DATA (заданная полем длины), а заголовок фрейма (размером в 9 октетов) исключается.

До тех пор пока все ресурсы не будут доставлены или соединение не будет закрыто клиентом, отправившим очень вежливый фрейм GOAWAY, оно продолжает функционировать аналогичным образом:

```
[  0.381] recv DATA frame <length=2563, flags=0x00, stream_id=13>
[  0.382] recv DATA frame <length=1491, flags=0x00, stream_id=13>
[  0.384] recv DATA frame <length=2581, flags=0x00, stream_id=13>
[  0.398] recv DATA frame <length=4072, flags=0x00, stream_id=13>
[  0.400] recv DATA frame <length=2332, flags=0x00, stream_id=13>
[  0.402] recv DATA frame <length=1491, flags=0x00, stream_id=13>
[  0.403] recv DATA frame <length=1500, flags=0x00, stream_id=13>
[  0.405] recv DATA frame <length=1500, flags=0x00, stream_id=13>
[  0.406] recv DATA frame <length=3644, flags=0x00, stream_id=13>
[  0.416] send HEADERS frame <length=250, flags=0x25, stream_id=15>
[  0.417] recv DATA frame <length=9635, flags=0x00, stream_id=13>
[  0.417] recv DATA frame <length=807, flags=0x00, stream_id=13>
[  0.419] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=33107)
[  0.419] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=13>
          (window_size_increment=33107)
[  0.420] recv DATA frame <length=16384, flags=0x00, stream_id=13>
[  0.420] recv DATA frame <length=369, flags=0x00, stream_id=13>
[  0.424] recv DATA frame <length=16209, flags=0x01, stream_id=13>
[  0.444] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=15>
          (window_size_increment=10420224)
[  0.546] recv (stream_id=15) x-frame-options: DENY
```

```
[  0.546] recv HEADERS frame <length=255, flags=0x04, stream_id=15>
[  0.546] recv DATA frame <length=1293, flags=0x00, stream_id=15>
[  0.546] recv DATA frame <length=2618, flags=0x00, stream_id=15>
[  0.547] recv DATA frame <length=3135, flags=0x00, stream_id=15>
[  0.547] send WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=34255)
[  0.547] recv DATA frame <length=10, flags=0x01, stream_id=15>
[  0.547] send GOAWAY frame <length=8, flags=0x00, stream_id=0>
```

Не видите фреймы WINDOW_UPDATE?

Если вы используете пример, отличный от Facebook (возможно, это даже ваш собственный сайт), вы можете удивиться отсутствию фреймов WINDOW_UPDATE. Возможно, выбранный вами сайт слишком мал и его можно загрузить целиком еще до отправки первого фрейма WINDOW_UPDATE.

Даже в примере с Facebook nghttp отправил фрейм WINDOW_UPDATE уже после 9 фреймов и 33 919 октетов (что намного меньше установленного нами предела 65 535). Если бы nghttp не отправил его, сервер мог бы продолжить загрузку данных.

Момент отправки фрейма WINDOW_UPDATE (после каждого фрейма DATA; когда объем данных приближается к пределу или периодически) определяется клиентом. Например, nghttp разрешает отправить его, когда задействовано больше половины размера окна управления потоком[a] (в данном примере 32 768 октетов). Именно поэтому здесь WINDOW_UPDATE отправляется после упомянутого выше фрейма DATA, имеющего размер 5572. До него общий размер составлял 28 347 октетов (ниже предела), а после 33 919 октетов (выше предела).

Если в качестве примера взять сайт Twitter, ответ для nghttp будет меньше чем 32 Кб (по крайней мере для запроса, не требующего входа в систему). Таким образом, nghttp не станет использовать фреймы WINDOW_UPDATE, и пример получится не таким интересным. Читатели могут экспериментировать с nghttp на своих сайтах, используя разные флаги (-w и -W) для разных начальных размеров окна[b].

[a] На https://github.com/nghttp2/nghttp2/blob/master/lib/nghttp2_helper.c вы можете найти функцию nghttp2_should_send_window_update.

[b] https://nghttp2.org/documentation/nghttp.1.html#cmdoption-nghttp-w.

7.2.2 Настройка управления потоком информации на сервере

В Apache вы можете установить размер окна управления потоком с помощью директивы H2WindowSize1:

```
H2WindowSize 65535
```

[1] https://httpd.apache.org/docs/2.4/mod/mod_http2.html#h2windowsize.

Такая опция есть и у других серверов. Например, NodeJS позволяет устанавливать эту директиву с помощью параметра `initialWindowSize`[1], а механизм сервлетов Jetty позволяет устанавливать его с помощью параметра `jetty.http2.initialStreamRecvWindow`[2]. Однако на момент написания этой книги многие другие серверы (такие как nginx и IIS) не позволяют вам как-либо взаимодействовать с этой директивой. Но в действительности вам вряд ли понадобится менять директиву по умолчанию, кроме случаев, когда разработчик хочет контролировать свой сервер полностью.

7.3 Приоритеты потоков

В этом разделе я расскажу о *приоритетах потоков*. HTTP/2 ввел в протокол концепцию приоритизации, которая позволяет клиенту придавать одним запросам важность относительно других. Как вы уже знаете, после загрузки страницы браузер начинает запрашивать ресурсы, необходимые для ее визуализации. Среди них высокий приоритет обычно имеют критические ресурсы, блокирующие рендеринг (например, CSS и любой блокирующий JavaScript), а низкий приоритет будет, например, у изображений или асинхронного JavaScript. Также сервер может использовать концепцию приоритизации при определении порядка отправки фреймов. Так, например, в первую очередь могут быть отправлены наиболее важные фреймы, ввиду чего клиент получит их раньше; также такой метод позволит обойти задержки и проблемы, связанные с управлением потоком или пропускной способностью.

Приоритет потока: подсказка или инструкция?

Приоритеты потоков устанавливаются запрашивающей стороной (например, клиентом), но итоговый порядок отправки фреймов определяет именно отправитель (например, сервер). Таким образом, приоритеты являются лишь предложениями или подсказками, которые отправитель может полностью игнорировать. Согласно спецификации «расстановка приоритета – это… всего лишь предложение от запрашивающей стороны»[a].

Так кто же все-таки должен определять приоритет: браузеры или серверы? Сложилось так, что браузеры всегда брали эту роль на себя. Они могли обрабатывать ограниченное количество HTTP/1.1-соединений, и поэтому им приходилось искать наиболее оптимальные варианты работы с ними. С появлением HTTP/2 ситуация перевернулась с ног на голову, и теперь за приоритизацию стал отвечать сервер. Такой вариант становится оправданным, если администратор веб-сайта осознанно настраивает веб-сервер для работы с конкретным сайтом. Однако, если не брать в расчет подобные случаи

---

[1] https://nodejs.org/api/http2.html#http2_settings_object.

[2] https://github.com/eclipse/jetty.project/blob/master/jetty-documentation/src/main/asciidoc/administration/http2/configuring-http2.adoc.

(к слову, не столь распространенные), веб-браузер все же лучше справляется с задачей приоритизации, нежели веб-сервер.

Я полагаю, большинство веб-серверов все же пользуются предлагаемыми клиентами подсказками касаемо расстановки приоритетов, и поэтому клиенты (например, веб-браузеры), скорее всего, и в дальнейшем будут определять приоритет. Конечно, у серверов может быть возможность предопределить данную настройку на своей стороне (см. раздел 7.3.4), однако я все же предполагаю, что в большинстве случае они продолжат следовать рекомендациям со стороны клиента.

Многие веб-серверы вообще отказываются от выполнения данной функции, поскольку реализовать ее довольно сложно. Но мне кажется, что те, кто все-таки может реализовать приоритизацию на своей стороне, обеспечивают своим пользователям неплохой прирост производительности. Поэтому я призываю вас выбирать веб-сервер (и веб-браузер) с умом!

[a] https://httpwg.org/specs/rfc7540.html#StreamPriority.

Устанавливать приоритет потока в HTTP/2 можно двумя методами:

- зависимостью потока;
- взвешиванием потока.

Данные приоритеты могут быть установлены во фрейме HEADERS. С помощью фрейма PRIORITY вы сможете поменять их в любое время.

7.3.1 Зависимости потоков

Один поток можно сделать зависимым от другого. В таком случае через него можно отправлять ресурсы только тогда, когда соединение не используется зависимым потоком. Один из подобных примеров приведен на рис. 7.3.

Согласно настройкам по умолчанию, главным является поток управления 0 (не показан на рис. 7.3). На рисунке показано, что main.css находится в зависимости с index.html и поэтому должен иметь наивысший приоритет; затем должен быть отправлен main.js и наконец image.jpg. Обычно сначала загружается index.html, а затем зависимые ресурсы, поэтому, возможно, устанавливать зависимость от файла документа HTML, как в этом примере, нет необходимости. Большой файл index.html может продолжать загружаться и во время выполнения других запросов, поэтому необязательно делать все запросы зависимыми от него.

Такая иерархия зависимостей не означает, что зависимые потоки блокируются для своих родительских потоков. Если main.css доступен для веб-сервера не сразу

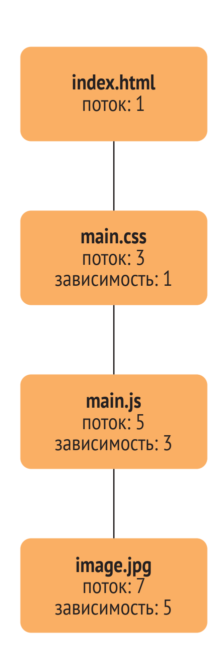


Рис. 7.3 Пример зависимости потока в HTTP/2

и должен быть получен, например, с внутреннего сервера, в это время сервер может отправить main.js, предполагая, что он доступен. Если ни один файл недоступен, во время ожидания вышеуказанных файлов будет отправлен image.jpg. Целью приоритизации потоков является эффективное использование соединения, и она не является механизмом блокировки.

Сервер может начать отправку image.jpg непосредственно во время получения им main.css и main.js. Когда эти файлы будут доступны для отправки, сервер может приостановить отправку image.jpg, отправить main.css и main.js, а затем снять image.jpg с паузы и продолжить отправку его оставшейся части. Существует и более простой альтернативный вариант, при котором сервер может завершать отправку image.jpg во время его запуска, пока остальные файлы встают в очередь согласно их степени готовности к отправке. Итак, выбор остается за сервером.

На рис. 7.4 показано, что родительские потоки могут иметь несколько зависимостей, и каждый поток может определять своего «родителя».

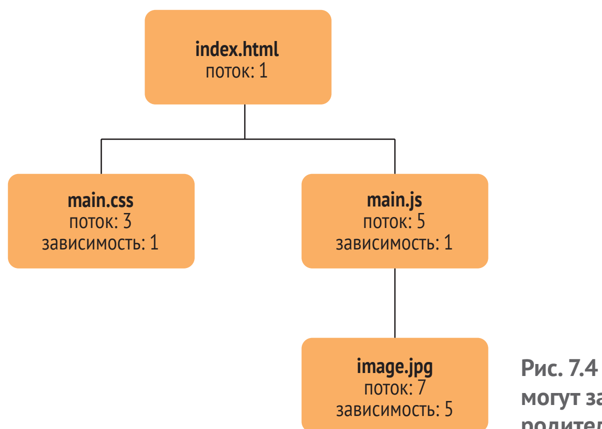


Рис. 7.4 Несколько потоков могут зависеть от одного и того же родительского потока

В этом примере и main.css, и main.js зависят от потока 1, а image.jpg зависит от потока main.js. В идеале если приоритет файла изображения ниже, чем у обоих критических ресурсов, то он будет зависеть от потоков CSS и JS, как показано на рис. 7.5, однако в HTTP/2 не поддерживается концепция множественных зависимостей.

Если бы эта концепция работала, загрузка файла начиналась бы только тогда, когда и main.css, и main.js закончили работу с соединением. Таким образом, множественные зависимости не входят в модель зависимостей HTTP/2 (хотя нечто похожее мы можем сделать посредством взвешивания потоков).

По мере того как ресурсы становятся доступными серверу для отправки, задача приоритизации потоков может усложниться. Кроме того, могут добавляться новые запросы, а старые завершаться на ходу. Зачастую с целью достижения оптимальной производительности серверы вынуждены давать приоритеты зависимостям несколько раз в течение всего жизненного цикла запроса. На ранних этапах реализации HTTP/2 Apache

обнаружил что отсутствие этих действий приводит к потере производительности[1].

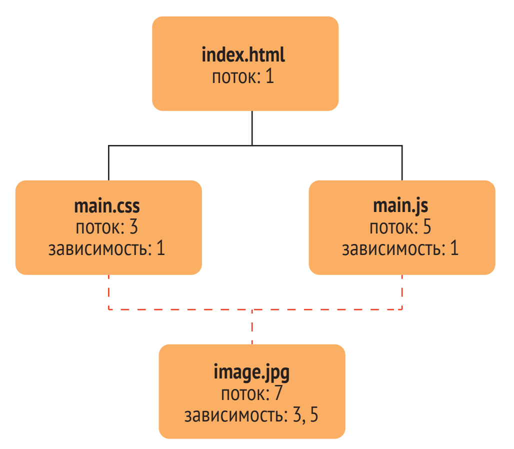


Рис. 7.5 Концепция множественных родительских зависимостей не поддерживается в HTTP/2

Потоки также могут вступать в исключающие зависимости, что означает, что один поток получает монопольный доступ к своей зависимости и остальные зависимости должны быть поставлены в зависимость от него. На рис. 7.6. изображен процесс добавления файла critical.css в систему зависимостей, где он зависит от потока 0. Слева процесс происходит без флажка исключительности, а справа с его использованием.

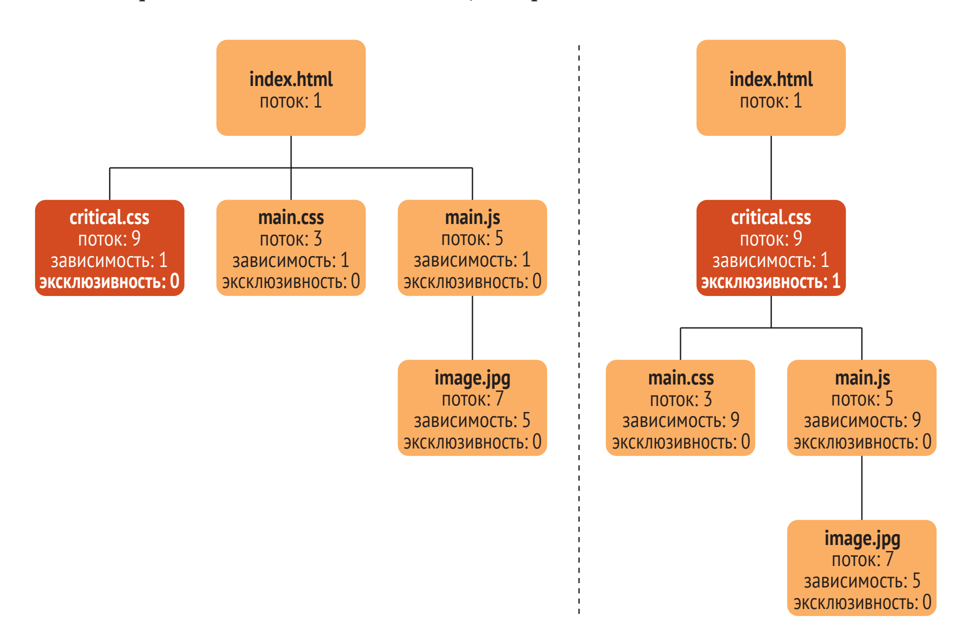


Рис. 7.6 Добавление новой зависимости critical.css с установленным флажком исключительности и без него

---

[1] http://icing.github.io/mod_h2/nimble.html.

Как видите, без этого флажка critical.css находится на том же уровне зависимости, что и main.css, и main.js, но после отметки его таким флажком, он получает приоритет и делает все остальные ресурсы зависимым от него. К слову, исходя из имени нашего ресурса (critical.css), это вполне то, что нужно.

7.3.2 Взвешивание потока

Взвешивание потоков – это еще одна концепция, позволяющая определять приоритеты потоков. Взвешивание применяется для определения приоритетов между двумя запросами, зависящими от одного и того же родительского ресурса. Взвешивание потоков допускает более сложные сценарии, чем взвешивание ресурсов на том же уровне зависимости. Например, на рис. 7.7 представлен сценарий загрузки critical.css с использованием взвешивания.

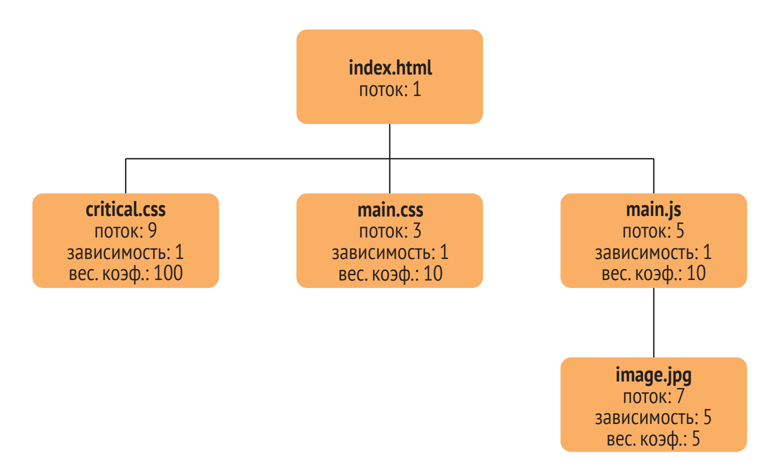


Рис. 7.7 Приоритеты потоков на основе взвешивания

На рисунке видно, что файл critical.css (оценка веса 100) должен получить в 10 раз больше ресурсов, чем main.css (оценка веса 10) и main.js (оценка веса также 10). Концепция взвешивания отличается от концепции зависимости, где ресурсы отмечаются исключающим флажком, однако все же они похожи. После полной загрузки critical.css файлы main.css и main.js получают по 50 % ресурсов, так как они находятся на одном уровне.

Весовой коэффициент 5 для image.jpg в данном сценарии не используется. Если main.js загружается раньше main.css (или если main.js еще не может быть отправлен), image.jpg получает 50 % ресурсов вместо main.js. Для того чтобы присвоить файлам CSS и JS больший весовой объем, чем изображениям, схему зависимостей можно построить несколько иначе, как на рис. 7.8.

Некоторые клиенты заранее создают фиктивные потоки с соответствующими приоритетами с помощью фрейма `PRIORITY`, а затем удер-

живают запросы. Таким образом, они упрощают приоритизацию. Концепция фиктивных фреймов `PRIORITY` была добавлена при ратификации HTTP/2[1]. Такой способ обеспечивает большую гибкость и позволяет использовать упрощенную модель приоритетов. На рис. 7.9 представлен пример его применения.

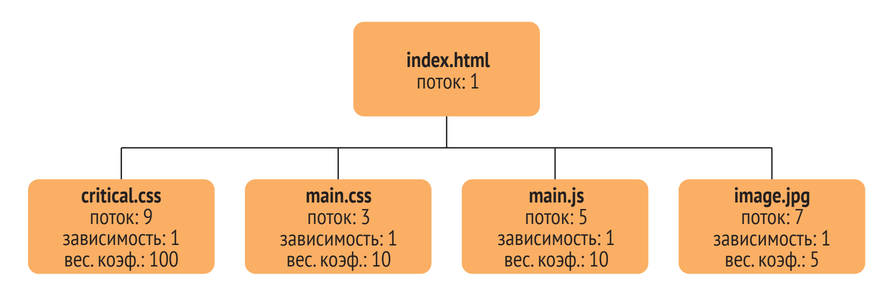


Рис. 7.8 Зависимости на основе взвешивания

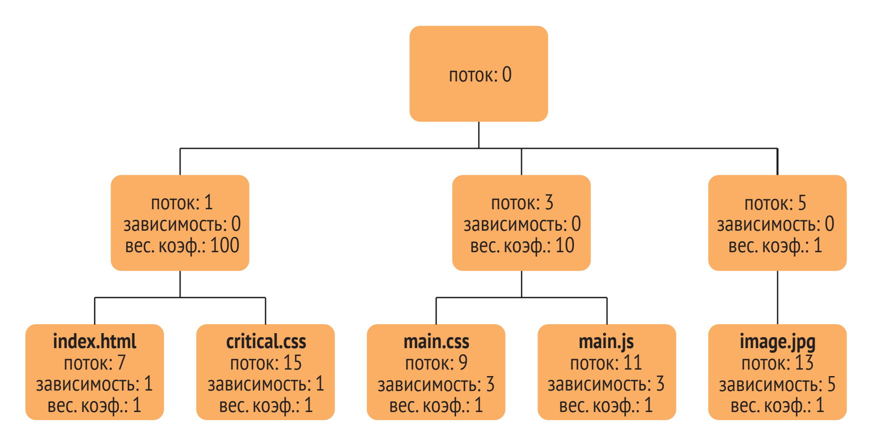


Рис. 7.9 Использование фиктивных потоков для удержания запросов в целях установки зависимости соответствующим образом

Фиктивные потоки могут использоваться только для определения приоритетов, но никак не для прямой отправки запросов. Такую ситуацию вы можете наблюдать, когда nghttp для облегчения расстановки приоритетов создает в начале соединения потоки 3, 5, 7, 9 и 11:

```
$ nghttp -nva https://www.facebook.com:443
[  0.041] Connected
The negotiated protocol: h2
[  0.093] recv SETTINGS frame <length=30, flags=0x00, stream_id=0>
          (niv=5)
          [SETTINGS_HEADER_TABLE_SIZE(0x01):4096]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16384]
          [SETTINGS_MAX_HEADER_LIST_SIZE(0x06):131072]
```

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/2014OctDec/0467.html.

```
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
[  0.093] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=10420225)
[  0.093] send SETTINGS frame <length=12, flags=0x00, stream_id=0>
          (niv=2)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
[  0.093] send SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
[  0.093] send PRIORITY frame <length=5, flags=0x00, stream_id=3>
          (dep_stream_id=0, weight=201, exclusive=0)
[  0.093] send PRIORITY frame <length=5, flags=0x00, stream_id=5>
          (dep_stream_id=0, weight=101, exclusive=0)
[  0.093] send PRIORITY frame <length=5, flags=0x00, stream_id=7>
          (dep_stream_id=0, weight=1, exclusive=0)
[  0.093] send PRIORITY frame <length=5, flags=0x00, stream_id=9>
          (dep_stream_id=7, weight=1, exclusive=0)
[  0.093] send PRIORITY frame <length=5, flags=0x00, stream_id=11>
          (dep_stream_id=3, weight=1, exclusive=0)
```

В результате создается дерево зависимостей, представленное на рис. 7.10. На нем поток 3 (с зависимым потоком 11) обладает наивысшим приоритетом, потоку 7 (с зависимым потоком 9) присваивается низший приоритет, и поток 5 получает средний приоритет.

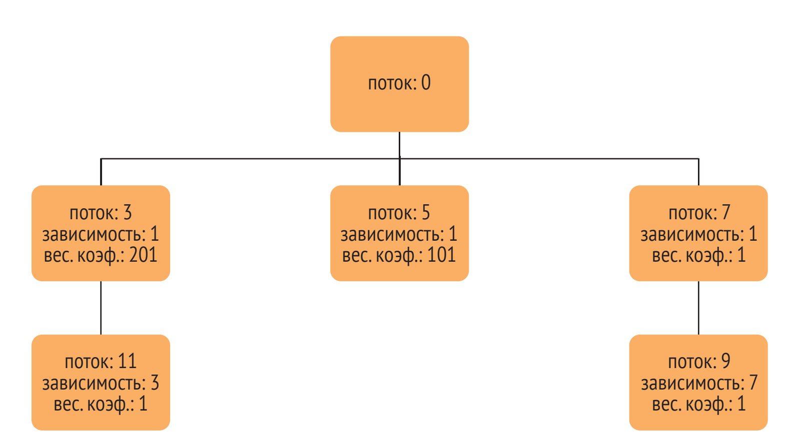


Рис. 7.10 Приоритеты потока nghttp

Любые запросы выполняются в зависимости от одного из этих потоков:

```
[  0.093] send HEADERS frame <length=43, flags=0x25, stream_id=13>
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=11, weight=16, exclusive=0)
          ; Open new stream
          :method: GET
          :path: /
```

```
:scheme: https
:authority: www.facebook.com
accept: */*
accept-encoding: gzip, deflate
user-agent: nghttp2/1.31.0
```

Данная настройка основана на исходном дереве зависимостей Firefox. Критические CSS и JavaScript зависят от потока 3, некритический JavaScript зависит от потока 5, а все остальные файлы зависят от потока 11. Обратите внимание, что в этом примере не используются потоки 7 и 9[1]. Таким образом, посредством создания зависимости у ресурсов от одних и тех же потоков вы сможете легко создать достаточно эффективную модель зависимостей.

7.3.3 Почему приоритизация – это так сложно?

Зачем нужны концепции *зависимости потоков и взвешивания*? Данный вопрос активно обсуждался после создания стандарта HTTP/2, а в SPDY, на котором он основывается, приоритизация осуществлялась на основе объема. Дело в том, что расстановка приоритетов – это довольно сложно, а использование зависимостей в совокупности с коэффициентами весового объема или их комбинации обеспечивает максимальную гибкость при расстановке приоритетов. Возможность создавать дополнительные потоки исключительно для целей приоритизации делает возможным воплощение самых разных реализаций.

На сегодняшний день поддержка приоритизации – это не обязательное требование, и многие реализации как на стороне клиента, так и на стороне сервера предпочитают отказываться от нее. Об этом я расскажу в следующем разделе. Как я уже отметил в разделе 6.2.4, способность эффективно обрабатывать приоритеты HTTP/2-потоков может стать еще одним ключевым отличием между реализациями браузера и сервера, хотя для большинства веб-пользователей и разработчиков технические детали могут так и остаться в тени.

С тех пор как был оформлен стандарт HTTP/2, поддержку приоритизации реализовали лишь немногие разработчики, и ни один из них еще не пришел к идеальной схеме ее реализации (об этом я расскажу позже). Ввиду этих фактов поступает множество просьб об упрощении реализации этого процесса[2]. Приведет ли это к каким-либо изменениям в HTTP/2 и будет ли приоритизация оптимизирована в будущих версиях (HTTP/3), еще неизвестно.

7.3.4 Приоритизация в веб-серверах и браузерах

Приоритизация HTTP/2 имеет большой потенциал и помогает использовать одиночное HTTP/2-соединение наиболее эффективным образом.

[1] https://nghttp2.org/documentation/nghttp.1.html#dependency-based-priority.
[2] https://lists.w3.org/Archives/Public/ietf-http-wg/2019JanMar/0073.html.

Такой вариант имеет преимущество перед вариантом HTTP/1.1 с шестью отдельными соединениями, где другой концепции относительной приоритизации не существует, и вы можете использовать одно из соединений. Однако на сегодняшний день приоритизация очень сложна, а ее поддержка ограничена. Несмотря на то что многие реализации как на стороне сервера, так и на стороне клиента поддерживают приоритизацию, лишь немногие из них предоставляют владельцу веб-сайта полный контроль над ней.

Установка приоритетов для веб-серверов

На момент написания этой книги мало что можно сказать однозначно относительно поддержки приоритизации серверами. Одни серверы поддерживают ее и предоставляют параметры конфигурации, другие поддерживают без предоставления параметров конфигурации, а некоторые не поддерживают вовсе. В табл. 7.1 представлена информация о поддержке приоритизации на популярных веб-серверах HTTP/2.

Таблица 7.1 Поддержка приоритизации на популярных веб-серверах HTTP/2

Сервер (и версия)	Поддержка приоритизации HTTP/2
Apache HTTPD (v2.4.35)	Приоритизация поддерживается, но можно настроить только приоритет Push[a]
IIS (v10.0)	Приоритизация не поддерживается[b]
nginx (v1.14)	Приоритизация поддерживается[c], но параметры конфигурации недоступны[d]
Node (v10)	Приоритизация поддерживается и может быть настроена[e]
nghttpd (1.34)	Приоритизация полностью поддерживается[f]

[a] https://httpd.apache.org/docs/2.4/mod/mod_http2.html#h2pushpriority.
[b] https://forums.iis.net/t/1233780.aspx.
[c] https://www.nginx.com/blog/http2-module-nginx/#prioritization.
[d] http://nginx.org/en/docs/http/ngx_http_v2_module.html.
[e] https://nodejs.org/api/http2.html#http2_http2stream_priority_options.
[f] https://nghttp2.org/blog/2014/04/27/how-dependency-based-prioritization-works/.

Веб-серверы практически не ссылаются на приоритизацию HTTP/2, поскольку предполагают, что, скорее всего, она не будет поддерживаться реализацией, и если это так, то ее настройка невозможна. Серверы, поддерживающие приоритизацию, предпочитают позволять клиенту самому указывать приоритеты в запросах, вместо того чтобы осуществлять их настройку на своей стороне.

Относительно новый веб-сервер Shimmercat реализует приоритизацию весьма интересным способом. Он позволяет изначально отвечать на запросы изображений с высоким приоритетом, а затем снижать его. Такой подход позволяет первым делом отправить первые несколько байтов данных, что дает браузеру понять размер изображения и другие метаданные, необходимые для создания макета страниц, а затем снизить приоритет для оставшейся части файла изображения.

Возможно, со временем количество веб-серверов, использующих подобные инновации и обеспечивающих владельцам сайта больший контроль, возрастет. Однако на данный момент большинство серверов ис-

пользует приоритеты, предложенные клиентом, или не поддерживают их вовсе.

Установка приоритетов для веб-браузеров

Поддержка приоритизации браузерами также осуществляется с переменным успехом. Документации по этой теме практически нет, но понять подход браузеров к приоритизации можно на примерах их использования. Для этого вы можете настроить Wireshark, как описано в главе 4, однако этот метод позволит вам перехватывать информацию только тех браузеров, которые экспортируют настройки ключа HTTPS (например, Chrome, Opera и Firefox для компьютеров). Наилучшим методом является запуск в подробном режиме сервера nghttpd и просмотр входящих сообщений. Также вы можете направить вывод сообщений в `grep` и отфильтровать среди них только нужные. Пользователи Windows без терминала Linux могут сделать то же самое с помощью `findstr` или `select-string`, если они используют PowerShell:

```
nghttpd -v 443 server.key server.crt | grep -E "PRIORITY|path|weight"
```

Затем необходимо создать фиктивный файл index.html в той же папке и загрузить ссылки на различные типы мультимедиа, чтобы получить представление о том, как каждый тип мультимедиа отправляется каждым браузером:

```
<html>
<head>
<title>This is a test</title>
<link rel="stylesheet" type="text/css" media="all" href="head_styles.css">
<script src="head_script.js"></script>
</head>
<body>
<h1>This is a test</h1>
<img src="image.jpg" />
<script src="body_script.js" /></script>
</body>
</html>
```

При этом не важно, существуют ли таблицы стилей, JavaScript или файлы изображений, на которые есть ссылка, на самом деле. Если они не существуют, будет даже проще, поскольку вы будете получать ответы 404 фрейма `HEADERS`, а не ответы фреймов `HEADERS` и `DATA`, затрудняющие прочтение.

Затем нужно подключиться к серверу (например, https://localhost) и просмотреть отправленные фреймы. Firefox (v62) отправляет фреймы аналогично клиенту nghttp, что неудивительно, поскольку тот основан на реализации Firefox:

```
[id=1] [  3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=3>
          (dep_stream_id=0, weight=201, exclusive=0)
[id=1] [  3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=5>
          (dep_stream_id=0, weight=101, exclusive=0)
```

```
[id=1] [  3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=7>
          (dep_stream_id=0, weight=1, exclusive=0)
[id=1] [  3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=9>
          (dep_stream_id=7, weight=1, exclusive=0)
[id=1] [  3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=11>
          (dep_stream_id=3, weight=1, exclusive=0)
[id=1] [  3.010] recv PRIORITY frame <length=5, flags=0x00, stream_id=13>
          (dep_stream_id=0, weight=241, exclusive=0)
[id=1] [  3.010] recv (stream_id=15) :path: /
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=13, weight=42, exclusive=0)
[id=1] [  3.033] recv (stream_id=17) :path: /head_styles.css
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=3, weight=22, exclusive=0)
[id=1] [  3.034] recv (stream_id=19) :path: /head_script.js
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=3, weight=22, exclusive=0)
[id=1] [  3.035] recv (stream_id=21) :path: /image.jpg
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=11, weight=12, exclusive=0)
[id=1] [  3.035] recv (stream_id=23) :path: /body_script.js
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=5, weight=22, exclusive=0)
```

В результате вы увидите, что Firefox добавил дополнительный поток 13 с объемом 241 (сверхсрочный поток?), используемый для исходного запроса, что сделало его более приоритетным, чем любой запрос CSS.

Chrome (v69) не использует фреймы `PRIORITY` в самом начале, как nghttp или Firefox, но устанавливает приоритет запросов при их отправке и добавляет зависимость от предыдущих потоков. Также он пользуется концепцией исключительной зависимости, создавая большой граф зависимостей:

```
[id=3] [112.082] recv (stream_id=1) :path: /
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=0, weight=256, exclusive=1)
[id=3] [112.101] recv (stream_id=3) :path: /head_styles.css
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=0, weight=256, exclusive=1)
[id=3] [112.101] recv (stream_id=5) :path: /head_script.js
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=3, weight=220, exclusive=1)
[id=3] [112.101] recv (stream_id=7) :path: /image.jpg
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=5, weight=147, exclusive=1)
[id=3] [112.107] recv (stream_id=9) :path: /body_script.js
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=0, weight=183, exclusive=1)
```

Эффективность использования исключающего бита все еще порождает вокруг себя споры[1]. Главный аргумент команды Chromium, по-

[1] https://bugs.chromium.org/p/chromium/issues/detail?id=651538.

видимому, состоит в том, что большинство запросов невозможно использовать до тех пор, пока не будет получен полный ресурс (основными исключениями являются HTML и прогрессивные JPEG-файлы), поэтому зачастую нет смысла «разбавлять» соединение несколькими ресурсами одновременно.

Opera (v59) придерживается того же способа, что и Chrome (так как тоже является браузером на основе Chromium). Однако Safari (v12.0) выполняет взвешивание на основе приоритетов и не использует потоковые зависимости (в отличие от Chrome!):

```
[id=9] [213.347] recv (stream_id=1) :path: /
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=0, weight=255, exclusive=0)
[id=9] [213.705] recv (stream_id=3) :path: /head_styles.css
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=0, weight=24, exclusive=0)
[id=9] [213.705] recv (stream_id=5) :path: /head_script.js
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=0, weight=24, exclusive=0)
[id=9] [213.706] recv (stream_id=7) :path: /image.jpg
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=0, weight=8, exclusive=0)
[id=9] [213.706] recv (stream_id=9) :path: /body_script.js
          ; END_STREAM | END_HEADERS | PRIORITY
          (padlen=0, dep_stream_id=0, weight=24, exclusive=0)
```

Реализация Edge (v41) наименее успешная из всех. Браузер не использует приоритеты потоков, поэтому каждый ресурс получает весовой приоритет по умолчанию, равный 16:

```
[id=4] [ 64.393] recv (stream_id=1) :path: /
[id=4] [ 64.616] recv (stream_id=3) :path: /head_styles.css
[id=4] [ 64.641] recv (stream_id=5) :path: /head_script.js
[id=4] [ 64.642] recv (stream_id=7) :path: /image.jpg
[id=4] [ 64.642] recv (stream_id=9) :path: /body_script.js
```

Как видите, в этом аспекте все браузеры придерживаются разных способов, что приводит к тому, что у одного и того же сайта, открытого в разных браузерах, может быть разная производительность. Исследователи уже провели более обширное тестирование различий между браузерами[1]. Вероятно, в будущем будут проводиться дополнительные исследования, а также появляться улучшения в этой области. HTTP/2 предоставляет инструменты для определения конкретных приоритетов, однако наилучшие способы их использования следует еще поискать.

[1] https://speakerdeck.com/summerwind/2-prioritization и https://www.researchgate.net/publication/324514529_HTTP2_Prioritization_and_its_Impact_on_Web_Performance.

7.4 Проверка совместимости с HTTP/2

Теперь, когда вы понимаете все тонкости HTTP/2, вы можете сравнить различные реализации как на стороне клиента, так и на стороне сервера.

7.4.1 Проверка совместимости сервера

H2spec[1] – это инструмент тестирования соответствия для реализаций HTTP/2. Он отправляет различные сообщения на сервер HTTP/2 и проверяет его соответствие спецификации. Вы можете загрузить версию данного инструмента для своего компьютера[2] и протестировать ваш сервер HTTP/2:

```
h2spec -t -S -h localhost -p 443
```

> **ПРИМЕЧАНИЕ** Если вы используете ненадежный сертификат (например, самозаверенный сертификат для localhost), чтобы игнорировать ошибки сертификата, воспользуйтесь параметром -k:
>
> ```
> h2spec -t -S -h -k localhost -p 443
> ```

Данный код запустит несколько тестов на вашем сервере и покажет их результаты:

```
$ ./h2spec -t -S -h localhost -p 443
Общие тесты для сервера HTTP/2
  1. Запуск HTTP/2
    ✓ 1: Отправка преамбулы к клиентскому соединению

  2. Потоки и мультиплексирование
    ✓ 1: Отправка фрейма PRIORITY в незанятом потоке
    ✓ 2: Отправка фрейма WINDOW_UPDATE в полузакрытом (удаленном) потоке
    ✓ 3: Отправка фрейма PRIORITY frame в полузакрытом (удаленном) потоке
    ✓ 4: Отправка фрейма RST_STREAM в полузакрытом (удаленном) потоке
    ✓ 5: Отправка фрейма PRIORITY в закрытом потоке

  3. Характеристики фреймов
    3.1. DATA
      ✓ 1: Отправка фрейма DATA
      ✓ 2: Отправка нескольких фреймов DATA
      ✓ 3: Отправка фрейма DATA с заполнением

    3.2. HEADERS
      ✓ 1: Отправка фрейма HEADERS
      ✓ 2: Отправка фрейма HEADERS с заполнением
      ✓ 3: Отправка фрейма HEADERS с приоритетом
…и т. д.
```

В табл. 7.2 представлены результаты проведенных мной тестов этого инструмента на некоторых популярных веб-серверах.

1 https://github.com/summerwind/h2spec.
2 https://github.com/summerwind/h2spec/releases.

Таблица 7.2 Соответствие популярных веб-серверов спецификации HTTP/2

Сервер (и версия)	Результаты тестов
Apache (v2.4.33)	146/146 (100 %)
nghttpd (v1.13.0)	145/146 (99 %)
Apache Traffic Server (v7.1.3)	140/146 (96 %)
CaddyServer (v0.10.14	137/146 (94 %)
HAProxy (v1.8.8)	136/146 (93 %)
IIS (v10)	119/146 (82 %)
AWS ELB	115/146 (79 %)
nginx (v1.13.9)	112/146 (77 %)

Исходя из предположения, что домашние страницы работают в инфраструктуре CDN, что, скорее всего, является неверным допущением, я провел аналогичные тесты для домашних страниц популярных сетей доставки контента. Результаты показаны в табл. 7.3.

Таблица 7.3. Соответствие популярных сетей CDN спецификации HTTP/2

CDN (и тестируемый сайт)	Результаты тестов
Fastly (www.fastly.com)	137/146 (94 %)
Google (www.google.com)	135/146 (92 %)
Cloudflare (www.cloudflare.com)	113/146 (77 %)
MaxCDN (www.maxcdn.com)	113/146 (77 %); обратите внимание, что тест 6.3.2 завис
Akamai (www.akamai.com)	107/146 (73 %)

Итак, наивысшего балла смог достичь только Apache, с чем я его, конечно же, поздравляю. Однако настолько ли большое значение имеет соответствие спецификации на 100 %? Вероятнее всего, нет. Снижение оценки иногда происходит из-за того, что серверы/CDN часто терпят неудачу при попытке обработать неправильные сообщения, которые не следовало бы отправлять в самом начале. Многие популярные серверы/CDN обрабатывают трафик HTTP/2 весьма успешно и без проблем, несмотря на то что у них есть погрешности в плане соответствия спецификации.

Взяв в качестве примера результаты ngix, вы увидите, что один из тестов сервер не проходит:

```
4.2. Размер фрейма
  ✓ 1: Отправка фрейма DATA длинной 2^14 октетов
  ✗ 2: Отправка фрейма DATA большого размера, превышающего
 SETTINGS_MAX_FRAME_SIZE
   -> Конечная точка ОБЯЗАНА отправить код ошибки FRAME_SIZE_ERROR.
      Ожидается: фрейм GOAWAY (Код ошибки: FRAME_SIZE_ERROR)
                 Фрейм RST_STREAM (Код ошибки: FRAME_SIZE_ERROR)
                 Соединение закрыто
    Фактическое: Фрейм WINDOW_UPDATE (длинна:4, метки:0x00, идентификатор потока:1)
```

Сервер ngix не может обработать большие фреймы `DATA` должным образом. Однако стоит отметить, что ни один клиент не должен отправлять такие фреймы. Среди прочих на сервере происходят ошибки состояния:

```
5. Потоки и мультиплексирование
  5.1. Состояния потока
    ✗ 1: неактивный: Отправка фрейма DATA
      -> Конечная точка ОБЯЗАНА рассматривать это как ошибку соединения типа
PROTOCOL_ERROR.
         Ожидается: фрейм GOAWAY (Код ошибки: PROTOCOL_ERROR)
                    Соединение закрыто
        Фактически: Timeout
    ✗ 2: неактивный: Отправка фрейма RST_STREAM
      -> Конечная точка ОБЯЗАНА рассматривать это как ошибку соединения типа
PROTOCOL_ERROR.
         Ожидается: фрейм GOAWAY (Код ошибки: PROTOCOL_ERROR)
                    Соединение закрыто
```

Опять же, nginx неправильно обрабатывает некорректно отправленные фреймы, когда поток находится в состоянии ожидания. Однако клиенты и не должны отправлять такие фреймы. Большинство других ошибок похоже на вышеописанные.

Если вы пишете сервер HTTP/2, инструмент h2 spec поможет вам проверить, насколько правильно ваш сервер реализует спецификацию. Однако реальность такова, что основные веб-серверы обходятся и не самыми безупречными реализациями. Internet всегда снисходительно относился к технологиям, и (в отличие от многих языков программирования) небольшие ошибки часто упускаются из виду. Тем не менее эти ошибки могут привести и к более неожиданным проблемам, поэтому вам может быть интересно узнать, как ведет себя ваш сервер. Когда я опубликовал указанную выше статистику в Twitter[1], несколько реализаций серверов приняли ее к сведению и попытались улучшить их соответствие.

7.4.2 Проверка совместимости клиента

Аналог H2spec существует и для клиентов (например, он подходит для тестирования браузеров)[2], однако во встроенных инструментах найти его вы не сможете. Данный инструмент должен быть скомпилирован из исходного кода. Я оставляю эту задачу читателю в качестве упражнения.

Резюме

- HTTP/2 реализует несколько сложных концепций, которые обсуждаются крайне редко, ввиду того, что многие люди концентрируются на концепциях более высокого уровня.
- Большинство операций низкого уровня, описанных в этой главе, не контролируется администраторами серверов или разработчиками веб-сайтов.

---

1 https://twitter.com/tunetheweb/status/988196156697169920.

2 https://github.com/summerwind/h2spec/pull/74.

- Существуют концепции состояния потока и схема состояний, которая показывает допустимые переходы между ними.
- HTTP/2 позволяет детально управлять потоками информации, не оставляя эту задачу TCP для управления на уровне соединения (как это происходит HTTP/1.1).
- HTTP/2 вводит понятие приоритетов потоков, позволяющих клиенту предлагать приоритет, который сервер будет использовать при ответе.
- Система приоритизации потоков HTTP/2 основана на концепциях зависимости и веса, которые можно использовать как вместе, так и отдельно.
- Все браузеры и серверы по-разному используют приоритеты потоков.
- Многие реализации HTTP/2 не полностью соответствуют спецификации.

8 Сжатие заголовков HPACK

В этой главе мы рассмотрим:

- историю сжатия данных;
- почему HTTP/2 необходима собственная техника сжатия заголовков;
- формат сжатия HPACK;
- распаковку закодированных заголовков HPACK;
- реализации HPACK на стороне клиента и сервера.

Следующая тема – сжатие заголовков. HTTP/1 позволял сжимать только HTTP-тела, а после появления HTTP/2 стало возможно сжатие HTTP-заголовков.

8.1 Для чего нужно сжатие заголовков?

В целом заголовки относительно малы, по сравнению с телами HTTP, однако, несмотря на это, их больше и зачастую они повторяются. Типичный HTTP/2-запрос `GET` от Chrome выглядит следующим образом:

```
:authority: www.example.com
:method: GET
:path: /url
:scheme: https
accept: text/html,application/xhtml+xml,application/xml;q=0.9,
        image/webp,image/apng,*/*;q=0.8
accept-encoding: gzip, deflate, br
```

```
accept-language: en-GB,en-US;q=0.9,en;q=0.8
upgrade-insecure-requests: 1
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_4) AppleWebKit/537
.36 (KHTML, like Gecko) Chrome/66.0.3359.139 Safari/537.36
```

В следующем запросе, отправленном на этот сервер, могут быть изменены только части, выделенные жирным шрифтом: метод запроса (`GET`) и путь (`/url`). Из 403 символов в данном коде уникальны для каждого запроса только 7. Вероятно, что для большинства запросов веб-страниц установлен метод `GET`, хотя веб-службы могут использовать и другие методы. Следовательно, изменен может быть только `:path` или URL, и в каждом запросе Chrome дублируются 399 символов, что очень затратно.

Ситуация усугубляется тем, что некоторые из этих заголовков довольно длинные, например заголовки `accept` и `user-agent`, показанные в предыдущем коде, а заголовки файлов куки могут быть еще длиннее. Ниже приведен заголовок запроса к Twitter (с обфусцированными значениями файлов куки):

```
:authority: twitter.com
:method: GET
:path: /
:scheme: https
accept: text/html,application/xhtml+xml,application/xml;q=0.9,
image/webp,image/apng,*/*;q=0.8
accept-encoding: gzip, deflate, br
accept-language: en-GB,en-US;q=0.9,en;q=0.8
cookie: _ga=GA1.2.123432087.1234567890; eu_cn=1; dnt=1; kdt=rmnAfbecvko4123
4oRYSzztq7n12345abcdABCD12; remember_checked_on=1; personalization_id="v1_k
0123451/EKaVeysDnuhKg=="; guest_id=v1%3A152383314680123456; ads_prefs="HBES
AAA="; twid="u=3374717733"; auth_token=12791876dfc0e57eae12345897b7940f55ac
7dfd; tfw_exp=1; csrf_same_site_set=1; csrf_same_site=1; lang=en; _twitter_
sess=BAh7CSIKZ12345678zonQWN0aW9uQ29udHJvbGxlcjo6Rmxhc2g6OkZsYXNo%250ASGFza
HsABjoKQHVzZWR7ADoPY3JlYXRlZF9hdGwrCPdpPx12345HaWQiJWY4%250AZGUwOGM3ZjRiYzJ
mYjRiAbCdEfGwNjIyZTk1Ogxjc3JmX2lkIiVkYjg3%2501234kZTVkMDdlMTAxMGI2YTgyZDFhN
TA0MmZiNQ%253D%253D--fd52ba1537f8fb9bf35dbd6080a6cd413edc6cd2; ct0=713653a0
6266b507960945523226bcc4; _gid=GA1.2.1893258168.1525002762; external_refere
r=1234567890w%3D|0|S381234567896Dak8Eqj76tqsc12345Lq4vYdCl5zxIvK6Q123vRkA%3
D%3D; app_shell_visited=1
referer: https://twitter.com/
upgrade-insecure-requests: 1
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_4) AppleWebKit/537
.36 (KHTML, like Gecko) Chrome/66.0.3359.139 Safari/537.36
```

Код состоит из 1278 символов, и опять же вполне вероятно, что при следующем запросе изменится только псевдозаголовок `:path`. Выходит, что при каждом запросе 1277 символов будут тратиться впустую. С ответами ситуация обстоит не лучше. На рис. 8.1 показан ответ Twitter.

Данный ответ состоит из 5804 символов (что откровенно нелепо). Большую часть объема занимает подробный (и большой) заголовок `content-security-policy`. Он является функцией безопасности, которая позво-

ляет веб-сайту сообщать браузеру, какой тип контента ему разрешено загружать на сайт[1].

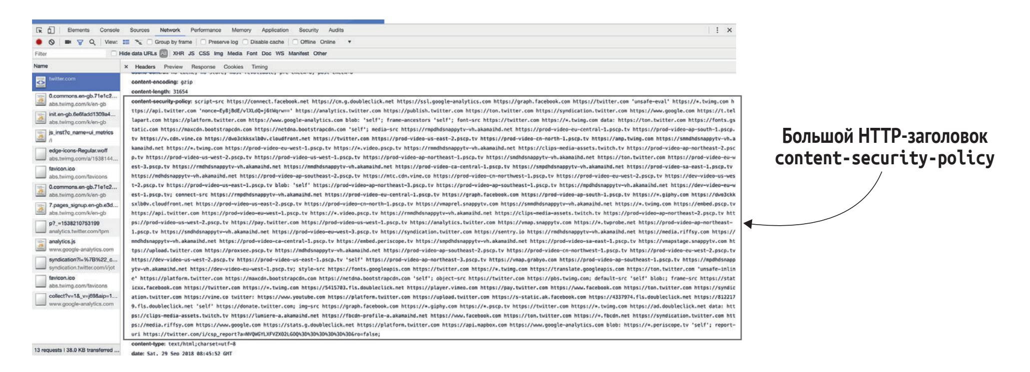


Рис. 8.1 Заголовок ответа Twitter

Программисты ненавидят повторения, именно это и стало причиной появления в HTTP/2 концепции сжатия HTTP-заголовков. Кроме того, оно стало ключевой частью протокола и было встроено даже в предшественника (SPDY). Сжатие и распаковка данных требует времени и вычислительной мощности, однако, по сравнению со временем, требуемым для отправки сетевых запросов, затраты не так велики. Именно поэтому сжатие данных перед их отправкой по сети почти всегда имеет смысл. Кроме того, HTTPS, например, требует шифрования, на которое затрачивается больше вычислительных ресурсов, чем на сжатие. Лучше сначала сжать данные, а затем зашифровать меньший объем.

8.2 Как работает сжатие

Чтобы разобраться в остальной части этого раздела, посвященного HTTP-сжатию, вам необходимо получить базовые знания о сжатии данных. Данная тема непроста, и при ее раскрытии я намеренно избегаю сложных математических аспектов. Несмотря на это, я постараюсь рассказать вам о многих деталях сжатия, чтобы читатели смогли понять, почему HTTP/2 реализует сжатие заголовков именно таким образом.

Иногда сжатие влечет за собой потери; некоторые данные можно отбросить, потому что они бывают не столь важны. Такой тип сжатия обычно применяется для мультимедиа: музыкальные файлы, изображения и видео возможно сжать без потери общего смысла данных. Однако, если сжать их слишком сильно, могут потеряться некоторые важные детали. Например, при чрезмерном сжатии изображения впоследствии его нельзя будет увеличить. При использовании *сжатия с потерями* необходимо соблюдать баланс между уменьшением размера и сохранением качества.

1 https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP.

HTTP-заголовки – это крайне важные данные, даже если они часто повторяются. К ним сжатие с потерями не применимо, несмотря на то что обычно оно дает лучший результат. *Сжатие без потерь* работает следующим образом: оно удаляет повторяющиеся данные, которые впоследствии могут быть восстановлены в несжатом виде. Существует три способа сжатия без потерь:

- использование таблиц подстановки;
- использование более эффективных методов кодирования;
- использование ретроспективного сжатия.

В следующих разделах я расскажу о каждом из этих методов подробнее.

8.2.1 Таблицы подстановки

Первый метод заключается в замене длинных повторяющихся частей данных ссылками. Распаковка происходит посредством замены этих ссылок исходным текстом из таблицы подстановки. Данный процесс может быть динамическим, но лучше всего он работает для данных одинаковой структуры. Взгляните на простой запрос `GET`:

```
:authority: www.example.com
:method: GET
:path: /url
:scheme: https
```

Он состоит из 64 символов. С помощью такой таблицы подстановки:

```
$1=:authority:
$2=:method:
$3=:path:
$4=:scheme:
```

вы можете закодировать этот текст так:

```
$1 www.example.com
$2 GET
$3 /url
$4 https
```

Полученный запрос сократился в размере на 40 % и теперь состоит из 39 символов. Однако в сжатую версию запроса может быть необходимо включить таблицу подстановки; это зависит от того, является она стандартной справочной таблицей известного формата или же динамической таблицей, сгенерированной для конкретного текста. Если включить таблицу в сжатую версию, общий размер запроса может вернуться к исходному количеству или даже стать больше, что противоречит самой концепции сжатия. Таблицы подстановки полезны только при наличии в запросе часто повторяющихся элементов.

В этом простом примере поисковые запросы представляют собой часто используемые имена HTTP-заголовков, поэтому для них можно ис-

пользовать предварительно согласованную статическую таблицу подстановки, в таком случае ее не нужно будет отправлять каждый раз. Она может быть дополнена таблицей динамической подстановки для дополнительных значений, которые будут использоваться, помимо предварительно согласованных. Например, домен www.example.com, вероятно, будет использоваться в последующих запросах, поэтому его можно будет добавить в динамический список для последующего использования ссылок.

8.2.2 Более эффективные методы кодировки

В данную категорию входит сразу несколько методов. Их общей идеей является то, что при выборе более эффективного способа представления сжимаемых данных их размер сокращается.

Например, пиксельные изображения могут быть основаны на 1-битном диапазоне пикселей (черный и белый). Если в изображении есть только красный и желтый, вы можете использовать 8-битную цветовую палитру, где вам понадобятся только два этих цвета. В качестве альтернативы вы можете использовать 1-битную палитру и указать что 0 – это желтый, а 1 – красный. Точно так же для текста вы можете использовать ASCII (7 бит), UTF-8 (8 бит для символов ASCII, 16 бит для часто используемых символов латиницы, 24 бита для большинства других распространенных символов и до 32 бит для остальных символов) или UTF-16 (16 бит для большинства символов и до 32 бит для остальных). UTF-8 (24 бита), больше подходит для английского языка, а UTF-16 (16 бит) – для кириллицы. Очень важно выбрать наиболее подходящий вам вариант, что позволить кодировать текст намного эффективнее.

Обратите внимание на кодирование с переменной длиной. Большинство методов кодирования работает по принципу кодирования фиксированного размера. Например, ASCII использует 7-битные символы, как показано в табл. 8.1.

Таблица 8.1 Часть кодов ASCII

Двоичный код	Символ
01000001	A
01000010	B
01000011	C
01000100	D
01000101	E
01000110	F
01000111	G
И т. д.	

Такая система удобна и проста, однако не очень эффективна, поскольку все буквы имеют код одинакового размера независимо от того, насколько часто они используются в языке. Самая распространенная буква в английском языке – E, затем идут T и A, и далее следуют менее рас-

пространенные вплоть до самых редких (X, J, Q и Z соответственно). Так почему же нужно относиться ко всем одинаково, если используются они по-разному?

Вместо фиксированного размера всех символов ASCII (7 бит) для текста на английском языке намного эффективнее использовать символы переменной длины; это значит, что для часто используемых символов будут применяться двоичные значения размером менее 7 бит, а для менее часто используемых букв более 7 бит. В некоторой степени такой метод применяет Unicode (UTF-8 и UTF-16), реализуя его посредством применения отдельных блоков символов (1–4 октета) в зависимости от частоты использования символов. Основная сложность заключается в распознавании границ между символами (потому что в этом формате они все имеют разные размеры).

Алгоритм Хаффмана (Huffman coding) выводит кодирование переменной длины на новый уровень. Согласно этой системе, каждому значению, в зависимости от частоты, с которой оно используется, присваивается уникальный код; таким образом, алгоритм гарантирует, что какой-либо код не является префиксом другого кода. Тема вычисления таких уникальных кодов выходит за рамки данной книги, однако в табл. 8.2 приведен небольшой пример (согласно кодам Хаффмана, основанным на частотном распределении букв в английском языке).

Таблица 8.2 Часть кодов ASCII с алгоритмом Хаффмана

Двоичный код	Алгоритм Хаффмана	Символ
01000001	1111	A
01000010	101000	B
01000011	01010	C
01000100	11011	D
01000101	100	E
01000110	01011	F
01000111	00001	G

В этом алгоритме ни один код не может быть представлен как начало другого. Например, для представления символов не используется код 0101, так как его можно спутать с какой-либо буквой, например началом кода буквы C или F. Если вы расшифровываете текст с самого начала, считайте его до тех пор, пока не найдете нужную букву; затем поищите другую и продолжайте так до полной расшифровки текста.

Таким образом, для того чтобы зашифровать слово *face*, вы можете использовать ASCII (4 × 7 бит = 28 бит) или алгоритм Хаффмана (5 + 4 + 5 + 3 = 17 бит). Даже в этом простом примере второй вариант позволяет значительно сократить размер кода. Соответственно, чем больше текст, тем больше эффективность.

Сжатие кодом Хаффмана – это расширение концепции таблиц подстановки, о которых я рассказывал в предыдущем разделе. Подобно им, таблицы Хаффмана могут быть определены заранее на основе известной структуры данных (например, текст на английском языке); также они

могут генерироваться динамически на основе зашифрованных данных или сочетать оба способа.

8.2.3 Ретроспективное сжатие

Механизм *ретроспективного сжатия* позволяет сократить размер кода посредством использования ссылок на повторяющийся текст относительно его текущего расположения. Особенно хорош данный тип сжатия для текстов HTML:

```
<html>
<body>
<h1>This is a title</h1>
<p>This is some text</p>
</body>
</html>
```

Этот текст может быть сжат следующим образом:

```
<html>
<body>
<h1> Заголовок </(-20,3)
<p>(-24,6)текст </(-19,2)
</(-58,4)
</(-73,4)
```

Каждый повторяющийся бит текста заменяется ссылкой, в которой указывается, как далеко декомпрессор может найти повторяющийся текст и сколько битов необходимо взять. Таким образом, ссылка (–20,3) говорит нам, что нужно вернуться на 20 символов назад и взять следующие три символа. Как видите, такой метод действительно удобен для HTML, в котором закрывающие теги повторяют открывающие (хотя, поскольку в этом примере мало тегов, таблица поиска может быть предпочтительнее). Вы также можете использовать этот тип сжатия для HTTP-заголовков, например `accept`:

```
accept: text/html,application/xhtml+xml,application/xml;q=0.9,
        image/webp,image/apng,*/*;q=0.8
```

В данном заголовке элементы `html`, `application`, `xml` и `image` повторяются, и поэтому их можно закодировать с помощью ретроспективного алгоритма. Вообще, компьютер никогда не работает со словами целиком; я использую слова в качестве примера, чтобы объяснение было более наглядным. Например, элемент `ml` есть и в `html`, и в `xml`, так что, возможно, предпочтительнее использовать именно его.

8.3 Сжатие HTTP-тел

Сжатие HTTP-тел обычно применяется для текстовых данных. В таком случае медиафайлы зачастую сжимаются предварительно (в зависимо-

сти от их формата) или не сжимаются вовсе. Например, JPEG – это особый формат сжатия изображений, ввиду чего веб-серверу не нужно сжимать их повторно; поэтому изображение больше не сжимается (и, напротив, может даже увеличиться в размере), и на обработку тела затрачивается больше времени. Все вышеописанные методы сжатия также хорошо подходят для текстового формата. Методы, используемые веб-серверами и браузерами (deflate, gzip и brotli), довольно похожи; они представляют собой варианты алгоритма на основе Deflate и для сжатия используют комбинацию различных методов. При выполнении запроса браузер сообщает серверу, какие алгоритмы сжатия он поддерживает, с помощью HTTP-заголовка `accept encoding`:

```
accept-encoding: gzip, deflate, br
```

Сервер выбирает один из этих алгоритмов и сжимает заголовок, а в ответе сообщает браузеру, какой алгоритм он использовал для сжатия ресурса:

```
content-encoding: gzip
```

Такая технология позволяет вводить новые алгоритмы сжатия (например, brotli), которые будут использоваться только при одновременной поддержке как со стороны клиента, так и со стороны сервера.

Сжатие на основе Deflate имеет один серьезный недостаток – недостаточную безопасность. Проблема состоит в том, что, исходя из длины кода, можно узнать содержимое и впоследствии повлиять на него. HTTP-тела могут содержать конфиденциальные данные (например, если ваше имя или номер учетной записи отображаются на странице), но все же большинство проблем безопасности связано с HTTP-заголовками, поскольку они содержат файлы куки и токены, используемые для аутентификации. Рассмотрим следующий запрос:

```
:authority: www.example.com
:method: GET
:path: /secretpage
:scheme: https
cookie: token=secret
```

Если бы вы знали значение токена (`secret`), то вы могли бы выдать себя за пользователя, которому он принадлежит. В данном случае сообщение зашифровано, и в идеале файл куки должен быть помечен как HttpOnly[1], чтобы JavaScript не мог его увидеть даже при внедрении его на страницу.

Однако при получении доступа к странице вы могли бы отправить следующие запросы с немного разными URL-адресами и измерить длину отправленного сообщения:

```
https://www.example.com/secretpage?testtoken=a
https://www.example.com/secretpage?testtoken=b
https://www.example.com/secretpage?testtoken=c
… и т. д.
```

[1] https://www.owasp.org/index.php/HttpOnly.

Методы сжатия на основе Deflate работают путем распознавания и замены повторяющихся шаблонов, и поэтому вы можете заметить, что один тест (`testtoken=s`) короче, чем другие тесты, потому что он повторяет первую часть реального файла куки (`token=secret`). Таким образом, вы узнаете первую букву значения токена и можете продолжать так до тех пор, пока не получите значение целиком:

```
https://www.example.com/secretpage?testtoken=sa
https://www.example.com/secretpage?testtoken=sb
https://www.example.com/secretpage?testtoken=sc
https://www.example.com/secretpage?testtoken=sd
https://www.example.com/secretpage?testtoken=se - короче!
https://www.example.com/secretpage?testtoken=sea
https://www.example.com/secretpage?testtoken=seb
https://www.example.com/secretpage?testtoken=sec - короче!
https://www.example.com/secretpage?testtoken=seca
https://www.example.com/secretpage?testtoken=secb
https://www.example.com/secretpage?testtoken=secc
… и т. д.
```

Такой процесс может показаться слишком долгим, однако написать соответствующий скрипт очень легко, и такой принцип уже применялся при реальной атаке, известной как CRIME (Compression Ratio Info-leak Made Easy)[1]. Она была совершена на SPDY, который для сжатия HTTP-заголовка использовал gzip.

8.4 Сжатие заголовка HPACK для HTTP/2

В виду проблем безопасности, выявившихся благодаря CRIME, в HTTP/2 пришлось использовать другой метод сжатия, который бы не был подвержен возникновению таких атак. Рабочая группа HTTP создала новую спецификацию под названием HPACK (не акроним), основанную на таблицах поиска и алгоритме Хаффмана, а (что особенно важно) не на ретроспективном сжатии.

HPACK[2] имеет свою спецификацию, отдельную от HTTP/2. Какое-то время велись дискуссии об их объединении, но в конце концов рабочая группа решила не делать этого. По большей части спецификация HTTP/2 не содержит подробностей о работе HPACK и ссылается на отдельную спецификацию[3]. Однако в ней говорится, что сжатие заголовков является частью HTTP/2 и что этот процесс структурирован (подробнее о том, почему, в разделе 8.4.2).

В отличие от многих других спецификаций HPACK не предназначена для расширения. В ней сказано[4]:

---

1 https://blog.qualys.com/ssllabs/2012/09/14/crime-information-leakage-attack-against-ssltls.

2 https://httpwg.org/specs/rfc7541.html.

3 https://httpwg.org/specs/rfc7540.html#HeaderBlock.

4 https://httpwg.org/specs/rfc7541.html#rfc.section.

Формат HPACK намеренно создан простым и негибким. Благодаря этому снижаются риски возникновения проблем совместимости или безопасности из-за ошибки в реализации. Механизмы расширения не определены; изменения формата возможны только путем полной замены.

На самом деле, я бы поспорил с утверждением о простоте HPACK. Я согласен с тем, что для интернет-спецификации он имеет действительно строгие рамки, что (как поясняется в цитате) было сделано из соображений безопасности). В будущем, скорее всего, появится новая версия HPACK (возможно, QPACK как часть QUIC; см. главу 9). Важно определить, как будет реализована эта версия (вероятно, при установке соединения настройка будет происходить иначе), но на данный момент HPACK определена довольно строго.

8.4.1 Статическая таблица HPACK

Статическая таблица HPACK включает в себя 61 наиболее распространенное имя HTTP-заголовков (и в некоторых случаях значения). В табл. 8.3 представлена ее часть. Полностью вы можете найти ее в спецификации HPACK[1].

Таблица 8.3 Часть статической таблицы HPACK

Индекс	Имя заголовка	Значение заголовка
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
5	:path	/index.html
6	:scheme	http
7	:scheme	https
8	:status	200
9	:status	204
10	:status	206
11	:status	304
12	:status	400
13	:status	404
14	:status	500
15	accept-charset	
16	accept-encoding	gzip, deflate
17	accept-language	
18	accept-ranges	
19	Accept	
...	…	…
60	Via	
61	www-authenticate	

[1] https://httpwg.org/specs/rfc7541.html#static.table.definition.

Данная таблица подходит как для запросов, так и для ответов. Она позволяет HTTP-сообщению эффективно сжимать наиболее распространенные имена и пары имя/значение. В результате заголовок

```
:method: GET
```

может быть сжат и представлен как ссылка на индекс 2.

А например, заголовок

```
:method: DELETE
```

в таблице не представлен, но его имя также можно сжать с ссылкой на индекс 2, а значение `DELETE` закодировать; т. е. с помощью этой таблицы мы можем сжимать не пару имя/значение целиком (например, `: method: GET`), а только имя (например, `:method`). Однако это работает в одностороннем порядке, поэтому найти значение, связанное с другим заголовком, вы не сможете. Например, в заголовке `header1: GET` значение `GET` не может быть представлено как ссылка на индекс 2.

Также заголовок `: method: DELETE` может быть сжат как ссылка на индекс 3 и закодированное значение `DELETE`. Подойдет любой из двух вариантов, так как он работает только с именем заголовка `: method`. В разделе 8.6 я расскажу об этом несколько подробнее.

8.4.2 Динамическая таблица HPACK

Помимо статической таблицы, на уровне соединения HPACK имеет и динамическую, которая начинается с индекса 62 (после статической) и продолжается вплоть до наибольшего возможного индекса, определенного значением `SETTINGS_HEADER_TABLE_SIZE` во фрейме `SETTINGS`. Если размер таблицы не указан, используется значение по умолчанию – 4096 октетов. В случае достижения максимального размера удаляются самые старые элементы списка. С целью упрощения этого процесса каждому элементу присваивается большее значение. Если запрос содержит два настраиваемых заголовка:

```
Header1: Value1
Header2: Value2
```

сначала элементом 62 станет заголовок `Header1`. После обработки `Header2`, `Header1` будет соответствовать индексу 63, а `Header2` – 62. Таким образом, позиция элемента заголовка в таблице не статична; его значение увеличивается по мере добавления новых заголовков как в текущем, так и в будущих запросах. Именно по этой причине требуются фреймы `HEADER` и `CONTINUATION`, поддерживающие целостность динамической таблицы. TCP гарантирует сохранение порядка в таких таблицах, и поэтому в HTTP/2 для каждого TCP-соединения создается отдельная динамическая таблица.

Данный процесс сложен, и лучше всего проиллюстрировать его реальным примером. Такой пример я привожу в разделе 8.5, поскольку для начала вам необходимо немного лучше разобраться в том, как работают ссылки на заголовки в динамических и статических таблицах.

8.4.3 Типы заголовков HPACK

Заголовки могут быть настроены (или не настроены) для добавления в динамическую таблицу. Существует четыре типа заголовков HPACK, которые я описываю в следующих разделах.

Литеральное представление поля заголовка

Литеральное представление поля заголовка (начинающееся со значения 1) – это прямой перенос значения в соответствии с таблицей (статической или динамической). Оно используется только для существующих в таблице имен заголовков и значений. Такой заголовок состоит из значения индекса таблицы, дополненного до 7 бит (как минимум). На рис. 8.2 представлен его формат.

0	1	2	3	4	5	6	7
1	Индекс (7+)						

Рис. 8.2 Формат литерального представления поля заголовка

Для больших чисел (7 или более бит) существует отдельный алгоритм[1]. Здесь я не буду рассказывать о нем подробно, лишь скажу, что он представляет собой присвоение каждому из 7 бит значения 1 и использование дополнительного(-ых) октета(-ов) для указания большего значения индекса.

Например, для кодирования `:method: GET` вы можете использовать индекс 2 (в двоичной системе `10`, а с начальными нулями `000 0010`). Если при сжатии литерального поля заголовка вы добавляете значение 1, вы получаете `1000 0010`, или `82` в шестнадцатеричном формате; такой результат вы можете видеть при просмотре трафика в Wireshark (рис. 8.3). В разделе 8.5 я объясняю, как просматривать нужные фреймы в Wireshark, и в качестве примера вы можете рассмотреть представленные там скриншоты.

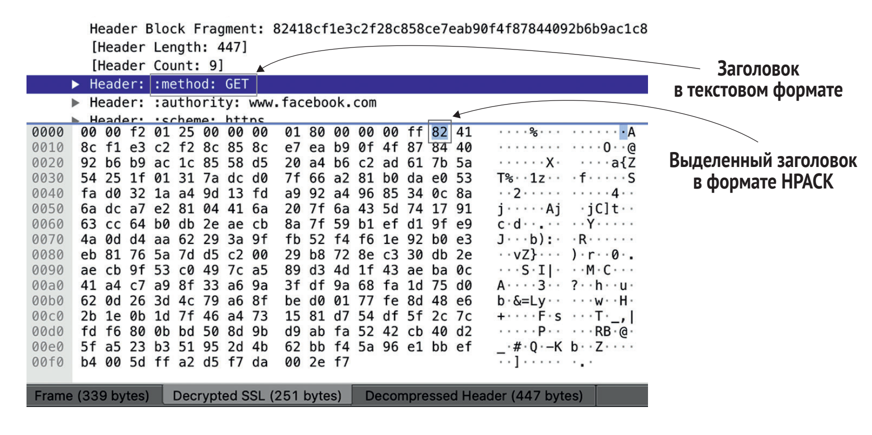


Рис. 8.3 HPACK-сжатие заголовка `:method: GET`

[1] https://httpwg.org/specs/rfc7541.html#integer.representation.

Литеральное поле заголовка с инкрементным типом индексирования

Литеральное поле заголовка с инкрементным типом индексирования (начинающееся с 01) используется, когда значение заголовка не представлено в таблице и его необходимо добавить в динамическую таблицу для дальнейшего использования.

Данный тип состоит из имени (которое может быть индексной ссылкой на имя заголовка, уже находящееся в таблице, или фактическим именем заголовка, которого еще нет в таблице) и значения заголовка.

Если используется индексированное имя заголовка (имя заголовка уже существует в таблице), биты после 01 (дополненные как минимум до 6 бит) определяют значение индекса, за которым следует само значение заголовка, как показано на рис. 8.4.

0	1	2	3	4	5	6	7
0	1	Индекс (6+)					
H	Длина значения (7+)						
Строка значения (октеты длины)							

Рис. 8.4 Литеральное поле заголовка с инкрементным форматом индексирования 1

Строка значения заголовка может быть закодирована (или не закодирована) по алгоритму Хаффмана (в зависимости от того, ускорит ли это процесс), а для 1-битного значения H устанавливается значение `1` (по Хаффману) или `0` (по ASCII).

На рис. 8.5 представлен реальный пример такого заголовка. Заголовок `authority:` имеет индекс `1`, поэтому он должен быть закодирован как `01000001` или `41` в шестнадцатеричном формате, за которым следует значение в кодировке Хаффмана (о которой я рассказываю в разделе 8.4.4).

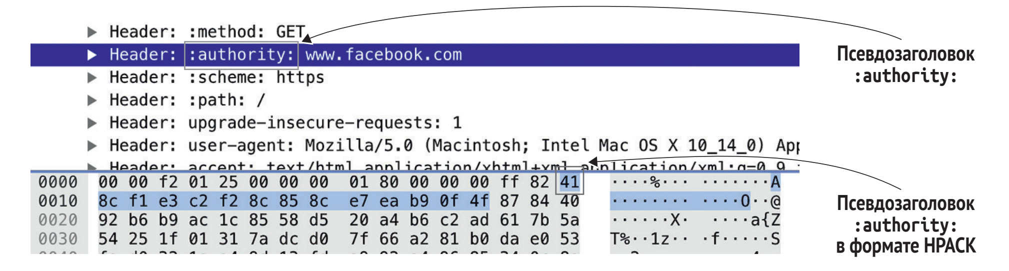


Рис. 8.5 HPACK-сжатие псевдозаголовка `:authority:` с новым значением

В противном случае для нового имени заголовка, не указанного в таблице поиска, после начального значения `01` 6 бит будут установлены со значением `0`, а имя и значение заголовка будет задано в виде пар длина/значение, как показано на рис. 8.6.

0	1	2	3	4	5	6	7
0	1	0					
H	Длина имени (7+)						
Строка имени (октеты длины)							
H	Длина значения (7+)						
Строка значения (октеты длины)							

Рис. 8.6 Литеральное поле заголовка с инкрементным форматом индексирования 2

Таким образом, первые 8 бит имеют значение `01000000` или `40`. На рис. 8.7 показан реальный пример, который начинается со значения `40`.

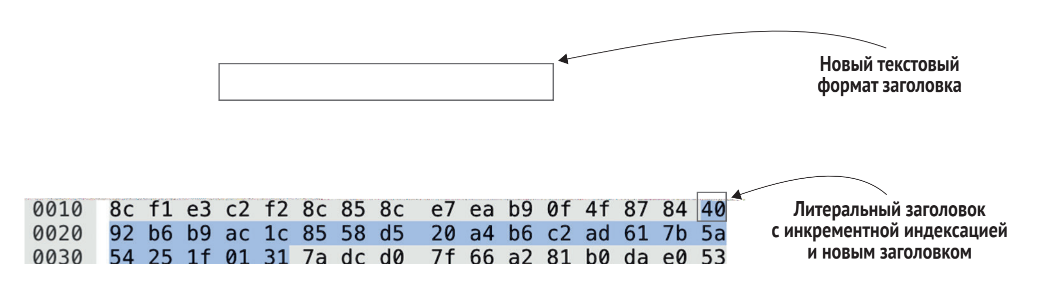


Рис. 8.7 Индексирование нового заголовка HPACK

Литеральное поле заголовка без индексирования

Литеральное поля заголовка без индексирования (начинающееся с 0000) используется для элементов, которые могут изменяться в запросах (например, `path`), из-за чего динамические таблицы могут стать неоправданно ресурсоемкими. Данный тип состоит из имени заголовка (которое может быть индексной ссылкой на имя поля в таблице или фактическим именем поля), однако имя и значение заголовка не сохраняются как элементы динамической таблицы. В зависимости от того, ссылается ли имя заголовка с помощью индекса или является фактическим, существует два формата такого типа кодирования; они представлены на рис. 8.8 и 8.9.

0	1	2	3	4	5	6	7
0	0	0	0	Индекс (4+)			
H	Длина значения (7+)						
Строка значения (октеты длины)							

Рис. 8.8 Формат литерального поля заголовка без индексирования 1

0	1	2	3	4	5	6	7
0	0	0	0	0			
H	Длина имени (7+)						
Строка имени (октеты длины)							
H	Длина значения (7+)						
Строка значения (октеты длины)							

Рис. 8.9 Формат литерального поля заголовка без индексирования 2

На рис. 8.10 показан пример HPACK-сжатия такого заголовка. Поскольку закодированный заголовок начинается с `00` в шестнадцатеричном формате (или `0000 0000` в двоичном), мы можем сделать вывод, что он относится к формату 2 и не ссылается на имя заголовка (`:path`) в таблице.

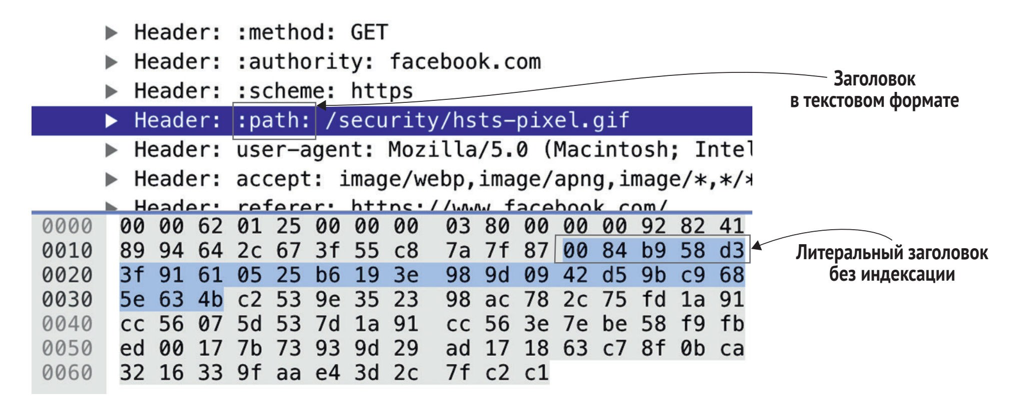


Рис. 8.10 Пример HPACK-сжатия заголовка без индексирования

Имя заголовка `:path` закодировано с помощью алгоритма Хаффмана и выглядит как `84 b9 58 d3 3f`, а значение заголовка (`/security/hsts-pixel.gif`), закодированное тем же методом, представлено в виде `91 61 05`… и т. д. Полностью сжатый заголовок выглядит следующим образом: `00 84 b9 58 d3 3f 91 61 05` и т. д., об этом я еще расскажу в разделе 8.4.4.

Такой формат кодирования кажется слегка странным и расточительным, поскольку заголовок `:path` уже есть в таблице (на позициях 4 и 5). Таким образом, заголовок мог быть закодирован в формате 1 и был бы на 5 октетов короче:

- формат 1 (индекс 5): `05 91 61 05` и т. д.;
- формат 2 (без индекса): `00 84 b9 58 d3 3f 91 61 05` и т. д.

В таких случаях Firefox использует первый вариант, а Chrome, по неизвестным причинам, – второй. Такая ситуация является ярким примером того, что кодировка от клиентов может быть совсем не такой, как мы ожидаем.

Литеральное поле заголовка строго без индексирования

Тип *литерального поля заголовка строго без индексирования* (начинающийся с 0001) аналогичен предыдущему типу, за исключением того, что при любых перекодировках (например, когда сервер действует как прокси между двумя реализациями HTTP/2) значение не добавляется в динамическую таблицу. Он создан для кодирования конфиденциальной информации (например, имени пользователя, пароля или всего сразу), которую вы хотели бы исключить из общего индекса HTTP-заголовка. Заголовок представляет собой способ обработки перекодировки, а также может быть самостоятельной кодировкой для заголовков в процессе их передачи.

Нужно ли хранить файлы куки в таблице HPACK?

Файлы куки являются конфиденциальными данными, и для их кодировки идеально подходит последний тип. Недостатком является разве что худшее сжатие файлов куки при последующих запросах. Файлы куки могут быть большими и повторяющимися, поэтому в идеале их *следует* сжимать.

В отличие от частичного ретроспективного сжатия при использовании HPACK необходимо угадать содержимое всего файла куки, прежде чем можно будет увидеть эффект (возможно, размер запроса будет меньше). Некоторые реализации (например, Firefox и nghttp)[a] для небольших файлов куки (менее 20 байт) используют только последний описанный тип; идея заключается в том, что содержимое больших файлов куки угадать труднее, поэтому здесь выигрыш от сжатия оправдан.

[a] https://github.com/nghttp2/nghttp2/blob/master/lib/nghttp2_hd.c.

Если отправитель решает использовать такой тип, настройки аналогичны настройкам предыдущих форматов «без индексирования» (рис. 8.11 и 8.12) в зависимости от того, ссылается ли заголовок с указателем на текущую таблицу или указывается полностью.

0	1	2	3	4	5	6	7
0	0	0	1	Индекс (4+)			
H	Длина значения (7+)						
Строка значения (октеты длины)							

Рис. 8.11 Формат литерального поля заголовка без индексирования 1

0	1	2	3	4	5	6	7
0	0	0	1	0			
H	Длина имени (7+)						
Строка имени (октеты длины)							
H	Длина значения (7+)						
Строка значения (октеты длины)							

Рис. 8.12 Формат литерального поля заголовка без индексирования 2

8.4.4 Таблица кодировки Хаффмана

Кодировка Хаффмана зависит от определения таблицы кодов, используемых для каждого символа в тексте. Для HPACK эта таблица определена в спецификации, поэтому и клиент, и сервер знают значения, которые следует использовать для кодирования и декодирования имен и значений заголовков. В табл. 8.4 представлена часть кодов Хаффмана для HPACK. Полную таблицу вы можете найти в спецификации HPACK[1].

Таблица 8.4 Часть таблицы кодировки Хаффмана для HPACK

<table>
<tr><th>Символ</th><th>Код ASCII</th><th>Код Хаффмана (двоичный)</th><th>Длина (в битах)</th></tr>
<tr><td>' '</td><td>(32)</td><td>|010100</td><td>[6]</td></tr>
<tr><td>'!'</td><td>(33)</td><td>|11111110|00</td><td>[10]</td></tr>
<tr><td>'"'</td><td>(34)</td><td>|11111110|01</td><td>[10]</td></tr>
<tr><td>'#'</td><td>(35)</td><td>|11111111|1010</td><td>[12]</td></tr>
<tr><td>'$'</td><td>(36)</td><td>|11111111|11001</td><td>[13]</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>'0'</td><td>(48)</td><td>|00000</td><td>[5]</td></tr>
<tr><td>'1'</td><td>(49)</td><td>|00001</td><td>[5]</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>'A'</td><td>(65)</td><td>|100001</td><td>[6]</td></tr>
<tr><td>'B'</td><td>(66)</td><td>|1011101</td><td>[7]</td></tr>
<tr><td>'C'</td><td>(67)</td><td>|1011110</td><td>[7]</td></tr>
<tr><td>'D'</td><td>(68)</td><td>|1011111</td><td>[7]</td></tr>
<tr><td>'E'</td><td>(69)</td><td>|1100000</td><td>[7]</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>'L'</td><td>(76)</td><td>|1100111</td><td>[7]</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>'T'</td><td>(84)</td><td>|1101111</td><td>[7]</td></tr>
<tr><td>…</td><td>…</td><td>…</td><td>…</td></tr>
</table>

[1] https://httpwg.org/specs/rfc7541.html#huffman.code.

Возвращаясь к предыдущему примеру, взгляните на заголовок:

```
:method: DELETE
```

Его можно сжать со ссылкой на индекс 2 и закодированным значением `DELETE`:

Буква	D	E	L	E	T	E
Код Хаффмана	1011111	1100000	1100111	1100000	1101111	1100000

В октетах заголовок будет иметь вид `1011 1111 1000 0011 0011 1110 0000 1101 1111 1000 00`, где последний дополнен единицами до `0011`. В шестнадцатеричном формате заголовок преобразуется в `bf 83 3e 0d f8 3f`, который и должен находиться перед длиной и флажком Хаффмана. В этом случае длина составляет 6 октетов, что в двоичном формате `110` или `000 0110` при дополнении до 7 бит. Добавив бит кодирования Хаффмана как 1 в начале октета длины (`1000 0110` или `86`), вы получите полностью закодированный заголовок `86 bf 83 3e 0d f8 3f`.

8.4.5 Скрипт кодирования по Хаффману

Кодирование и декодирование HPACK по Хаффману легко поддается автоматизации. В следующем листинге показана одна из таких реализаций на Perl. Обратите внимание, что в целях экономии места показана только часть таблицы Хаффмана. Полный список доступен на странице этой книги на GitHub[1].

Листинг 8.1 Простой кодировщик HPACK по Хаффману

```
#!/usr/bin/perl

use strict;
use warnings;

#Прочтите строку для преобразования из командной строки.
my ($input_string) = @ARGV;

if (not defined $input_string) {
  die "Need input string\n";
}

#Установите и заполните хеш-переменную всеми значениями поиска Хаффмана.
#Обратите внимание, что в этом простом примере используются только пригодные
#для вывода на печать значения.
my %hpack_huffman_table;

$hpack_huffman_table{' '} = '010100';
$hpack_huffman_table{'!'} = '1111111000';
$hpack_huffman_table{'\"'} = '1111111001';
$hpack_huffman_table{'#'} = '111111111010';
…и т.д.
```

[1] https://github.com/bazzadp/http2-in-action.

```
$hpack_huffman_table{'}'} = '11111111111101';
$hpack_huffman_table{'~'} = '1111111111101';

#Установите двоичную переменную строки
my $binary_string="";

#Разделите строку ввода по символам.
my @input_array = split(//, $input_string);

#Для каждого входного символа найдите строку в хеш-таблице Хаффмана
#и добавьте его в двоичную строковую переменную.
foreach (@input_array) {
  $binary_string = $binary_string . $hpack_huffman_table{$_};
}

#Дополните двоичную строку, чтобы она делилась на 8.
while (length($binary_string) % 8 != 0) {
        $binary_string = $binary_string . "1";
};

#Рассчитайте длину, разделив на 8.
my $string_length = length($binary_string)/8;

#Подобная простая реализация не обрабатывает большие строки
#(оставлено в качестве упражнения для читателя).
if ($string_length > 127) {
        die "Error string length > 127 which isn't handled by this
     program\n";
}

#Установите самый старший бит (128), чтобы указать, что используется кодировка
#Хаффмана.
#Укажите длину
#(опять же, эта простая версия предполагает длину в 7 бит.)
printf("Huffman Encoding Flag + Length: %x\n",128+$string_length);

#Пройдитесь по каждому 4-битному значению и преобразуйте их в шестнадцатеричный
#формат.
for(my $count=0;$count<length($binary_string);$count = $count + 4) {
        printf("%x",oct("0b" . substr($binary_string,$count,4)));
}
printf ("\n");
```

В данном листинге представлена HPACK-кодировка по Хаффману в шестнадцатеричном формате. Взгляните на пример:

```
$ ./hpack_huffman_encoding.pl DELETE
Huffman Encoding Flag + Length: 86
Huffman Encoding Value        : bf833e0df83f
```

Вы можете использовать аналогичный скрипт и для декодирования строк. Также его можно улучшить, чтобы он принимал список заголовков и обрабатывал состояние динамической таблицы. Я оставляю обе эти задачи в качестве упражнений, которые читатели должны выполнить на выбранном ими языке.

Несмотря на то что кодирование Хафманна довольно сложно выполнять вручную, его легко можно реализовать в коде и эффективно использовать для компьютеров. Вряд ли вы захотите кодировать или декодировать заголовки вручную, как я сделал в этой главе.

8.4.6 Почему кодирование Хафманна подходит не во всех случаях

В некоторых случаях кодирование по Хаффману может значительно увеличить длину строки, в сравнении, например, с простым ASCII. При кодировке `delete` с помощью ASCII, вы можете сразу перейти к шестнадцатеричной системе (потому что каждый код ASCII имеет длину 1 октет):

Буква	D	E	L	E	T	E
Шестнадцатеричный код ASCII	44	45	4C	45	54	45

Как видите, версия, закодированная с помощью ASCII, имеет длину 6. При кодировании по Хаффману, если установить значение флажка `0`, вы ставите перед заголовком `06` и получаете результат в виде `06 44 45 4C 45 54 45`.

В подобном случае выбранный вами алгоритм кодировки не имеет никакого значения – длина обоих результатов будет равняться 7 октетам. Несмотря на то что все коды Хаффмана, используемые в этом примере, являются 7-битными (в сравнении с 8-битными кодами ASCII), заполнение необходимо для округления до полных октетов, поэтому коды в конечном итоге имеют одинаковый размер. Для других заголовков кодировка ASCII может быть компактнее, как если бы в заголовке использовались символы из таблицы Хаффмана (длиннее 8 бит), которые относятся к редко используемым. По этой причине спецификация HPACK оставляет использование алгоритма Хаффмана на усмотрение пользователя, а также позволяет выбирать другие методы для новых заголовков. Можно использовать любую кодировку, позволяющую выразить значение в наименьшем количестве октетов.

Однако зачастую кодирование Хаффмана более эффективно, чем ASCII. Частично это объясняется тем фактом, что для ASCII требуется только 7 бит, но используется полный 8-битный октет, поэтому 1 бит тратится впустую для каждого значения, закодированного в ASCII. Кодирование Хаффмана позволяет использовать кодирование переменной длины, поэтому теоретически биты не тратятся зря. Однако при кодировании с переменной длиной редко используемые символы могут занимать более 8 бит, поэтому их лучше кодировать в ASCII. Такие значения по определению должны встречаться не часто (при условии, что таблица кодирования HPACK Хаффмана отражает реальное использование). Наконец, поиск в статических или динамических таблицах всегда более эффективен, чем кодирование по Хаффману или ASCII.

8.5 Практические примеры сжатия HPACK

В предыдущих разделах я изложил очень много теоретической информации. В этом же разделе я приведу вам несколько реальных примеров, которые проиллюстрируют все вышесказанное. Большинство инструментов, поддерживающих HTTP/2, занимаются сжатием HPACK самостоятельно, незаметно для пользователя. Поэтому сейчас вернитесь в Wireshark и просмотрите необработанные данные, отправляемые по сети (и расшифрованную версию этих данных). Я рассказывал о Wireshark в главе 4, там вы можете найти необходимую информацию для работы с ним.

Допустим, вы осуществляете прослушивание трафика HTTP/2; обратите внимание на заголовки. На рис. 8.13 представлен пример запроса к Facebook, где первым фреймом является `HEADERS` (вторая строка в окне).

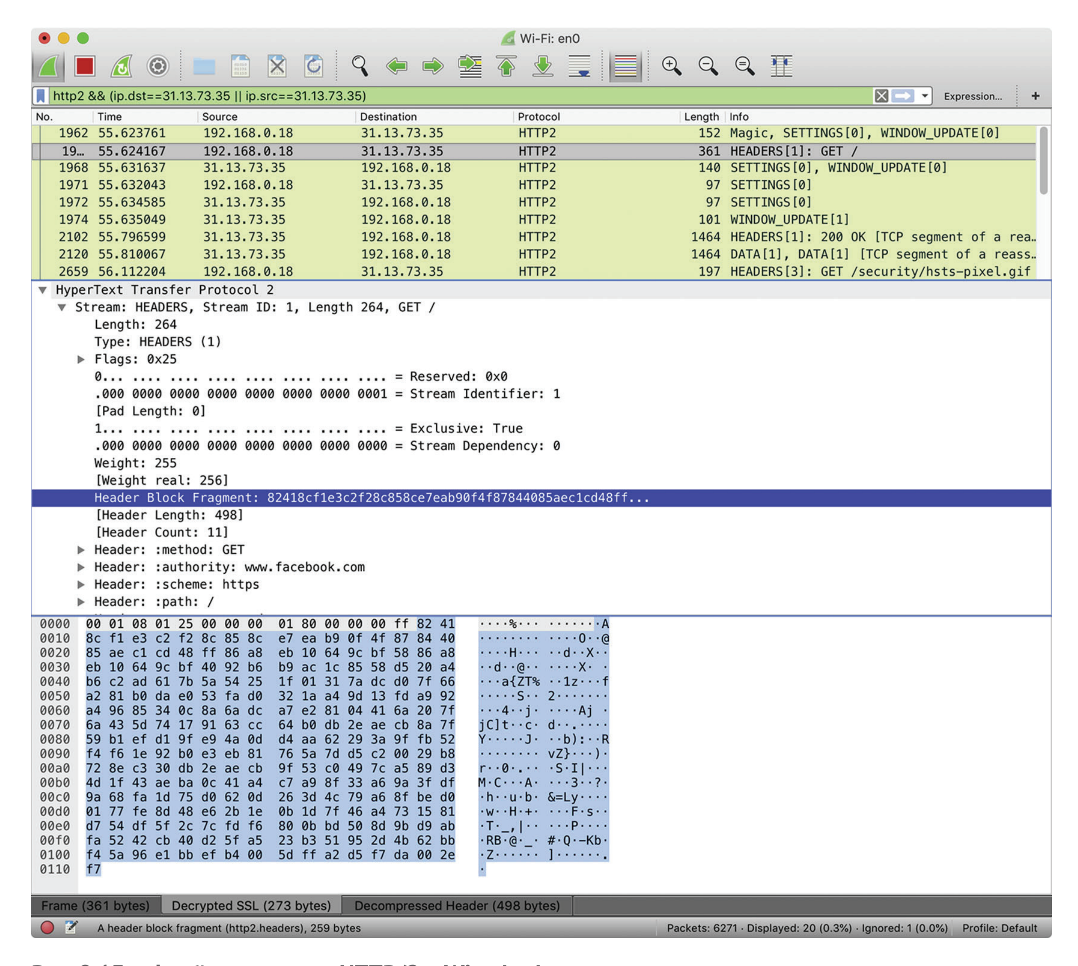


Рис. 8.13 Фрейм заголовка HTTP/2 в Wireshark

В нижней части рисунка вы можете увидеть, что размер фрейма Wireshark (здесь используется в значении «пакет», не путайте с фреймом HTTP/2) составляет 361 байт. После расшифровки его размер становится 273, а после распаковки возрастает до 489. Здесь даже для первого фрейма мы получаем хороший процент сжатия. Экономия составляет 45 % (273 / 498 = 55 %), хотя некоторая ее часть теряется при шифровании.

Если не брать в расчет детали фрейма HTTP/2 (такие как тип заголовка, флажки, вес и т. д.), закодированный заголовок в шестнадцатеричном формате выглядит следующим образом:

```
82 41 8c f1 e3 и т. д.
```

С помощью полученных знаний расшифруем этот заголовок. Здесь мы имеем дело с кодировкой Хаффмана переменной длины, ввиду чего рассматривать заголовок в виде октетов бессмысленно. Предпочтительнее преобразовать его в двоичный формат:

```
82418cf1e3… = 1000 0010 0100 0001 1000 1100 1111 0001 1110 0011…
```

На этом этапе заголовок становится читаемым и понятным. Вы знаете, что есть четыре типа заголовков:

- *литеральное представление поля заголовка* (начинается с `1`);
- *литеральное поле заголовка с инкрементным типом индексирования* (начинается с `01`);
- *литеральное поле заголовка без индексирования* (начинается с `0000`);
- *литеральное поле заголовка строго без индексирования* (начинается с `0001`).

Данный блок заголовка начинается с `1`, поэтому он относится к первому типу. Следующие 7 бит ссылаются на индекс таблицы `2`, что соответствует `:method: GET` – вашему первому несжатому заголовку! Этот заголовок хранился в 1 октете (`82`), а для кодирования каждого символа в ASCII требуется 12 октетов. Таким образом, экономия налицо. Итак, расшифруем первые 8 бит:

~~1000 0010~~ `0100 0001 1000 1100 1111 0001 1110 0011…`

Следующая часть начинается с `01`, поэтому она относится ко второму типу. Поскольку за ним не следуют 6 нулей, делаем вывод, что 6 байт ссылаются на имя индекса. На рис. 8.14 представлен инкрементный формат с двоичными значениями.

Биты								Реальные значения
0	1	2	3	4	5	6	7	
0	1	Индекс (6+)						0100 0001
H	Длина значения (7+)							1000 1100
Строка значения (октеты длины)								1111 0001 … и т. д.

Рис. 8.14 Формат 1 литерального поля заголовка с инкрементным типом индексирования

Теперь расшифруйте первые октеты следующего заголовка (0100 0001).

Если отбросить 01, получим индекс 00 0001 или 1, что в таблице соответствует заголовку :authority. Расшифруем первые 16 бит:

```
~~1000 0010 0100 0001~~ 1000 1100 1111 0001 1110 0011 1100…
```

Теперь вам нужно найти значение для этого заголовка :authority. Первым символом следующего октета является 1, поэтому вы знаете, что предстоящее значение является значением, закодированным по Хаффману, и что длина – это остаток от этого октета, или 000 1100 = 8 + 4 = 12 октетов:

```
1000 1100 Строка кода Хаффмана с длиной 12.
```

Затем расшифруем 24 бита:

```
~~1000 0010 0100 0001 1000 1100~~ 1111 0001 1110 0011 1100…
```

Следующие 12 октетов будут выглядеть следующим образом:

```
1111 0001 1110 0011 1100 0010 1111 0010 1000 1100 1000 0101
1000 1100 1110 0111 1110 1010 1011 1001 0000 1111 0100 1111
```

Далее просто читайте каждый бит до тех пор, пока не найдете уникальное значение из таблицы Хаффмана. Сейчас с целью экономии времени я сделаю это за вас (для программы это очень легко, но для человека намного сложнее:

- 1111000 уникальное значение, идентифицируется как w;
- 1111000 уникальное значение, идентифицируется как w;
- 1111000 уникальное значение, идентифицируется как w;
- 010111 уникальное значение, идентифицируется как . .
  и т. д.;
- 00100 уникальное значение, идентифицируется как c;
- 00111 уникальное значение, идентифицируется как o;
- 101001 уникальное значение, идентифицируется как m;
- 111 служит для заполнения последнего октета.

Таким образом, полное значение – www.facebook.com, и в сочетании с именем заголовка (:authority) мы получаем:

```
:authority: www.facebook.com
```

Данный заголовок заносится в динамическую таблицу с индексом 62. Индекс 62 является ближайшим свободным, поскольку статическая таблица заканчивается индексом 61.

Проведя те же манипуляции с оставшейся частью фрейма HEADERS, вы получите динамическую табл. 8.5.

Как видите, заголовок :authority: имеет индекс 69. Следовательно, порядок заголовков очень важен и может не совпадать с заголовками, показанными в инструментах разработчика: например, Chrome упорядочивает заголовки в алфавитном порядке в инструментах разработчика, а отправляет в другом порядке. Также обратите внимание, что в таблицу были добавлены не все заголовки, поскольку, например, псевдозаголов-

ки :scheme: https и :path: / уже есть в статической таблице (с индексами 7 и 4 соответственно).

Таблица 8.5 Динамический заголовок после получения первого фрейма HEADERS

Индекс	Заголовок	Значение
62	accept-language	en-GB,en-US;q=0.9,en;q=0.8
63	accept-encoding	gzip, deflate, br
64	Accept	text/html,application/xhtml+xml, application/xml;q=0.9,image/webp, image/apng,*/*;q=0.8
65	user-agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36
66	upgrade-insecure-requests	1
67	cache-control	no-cache
68	Pragma	no-cache
69	:authority	www.facebook.com

Заголовки первого запроса должны быть занесены в динамическую таблицу, поэтому, если их там не будет, вы не получите хороший процент сжатия. Последующие запросы смогут использовать эти значения и улучшить степень сжатия, как показано на рис. 8.15.

Следующий полученный фрейм HEADERS я пропущу и перейду к следующему отправленному HEADERS. На основе IP-адресов источников и пунктов назначения вы можете увидеть маршрут каждого запроса. Отправленные и полученные заголовки обрабатываются отдельно, но одинаково. Здесь я большее внимание уделяю отправляющей стороне клиента; на серверной стороне отправка происходит таким же образом, но с отдельной управляемой таблицей динамических заголовков.

Второй отправляемый запрос начинается со значения 82. С этим мы уже сталкивались, и вы знаете, что имя заголовка до :method: GET не сжато/не закодировано, поэтому я не буду повторяться. Второй заголовок несколько интереснее. 41 – это заголовок :authority, за которым следует значение в кодировке Хаффмана (facebook.com). Обратите внимание, что этот домен отличается от первого запроса (facebook.com – www.facebook.com), поэтому ранее сохраненное динамическое значение не может быть использовано повторно; вместо этого значение следует отправить. В этом примере также на практике показано слияние соединений, поскольку для каждого HTTP/2-запроса (даже для другого домена) отдельное соединение не требуется. Чтобы получить дополнительную информацию о слиянии соединений, обратитесь к главе 6.

Следующие несколько заголовков также довольно интересны. На рис. 8.16 я демонстрирую заголовок user-agent.

Как видите, для первого запроса требуется заголовок user-agent[1], он должен быть представлен полностью, и для его отправки используется 94 октета. Во втором запросе заголовок не изменился, поэтому в целях

[1] Если вам интересно, почему заголовок такой длинный, ознакомьтесь с https://webaim.org/blog/user-agent-string-history/.

экономии он может быть отправлен в двух октетах (`c2`). Таким образом, экономия даже больше чем при сжатии текстового заголовка HTTP/1.1 ASCII, для которого понадобился бы 131 октет.

Чтобы увидеть, как `c2` преобразуется в заголовок `user-agent`, сначала вам нужно понять, что был добавлен новый заголовок (`:authority: facebook.com`) и что он был бы сохранен в динамической таблице, из-за чего индексы сдвинулись вверх.

Теперь вы можете распаковать заголовок `user-agent`:

```
c5 = 1100 0010
```

Первое значение `1` – это литеральное соответствие имени и значения заголовка, а `100 0010` переводится в 66, и (как показано в табл. 8.6) это значение является сохраненным заголовком `user-agent`.

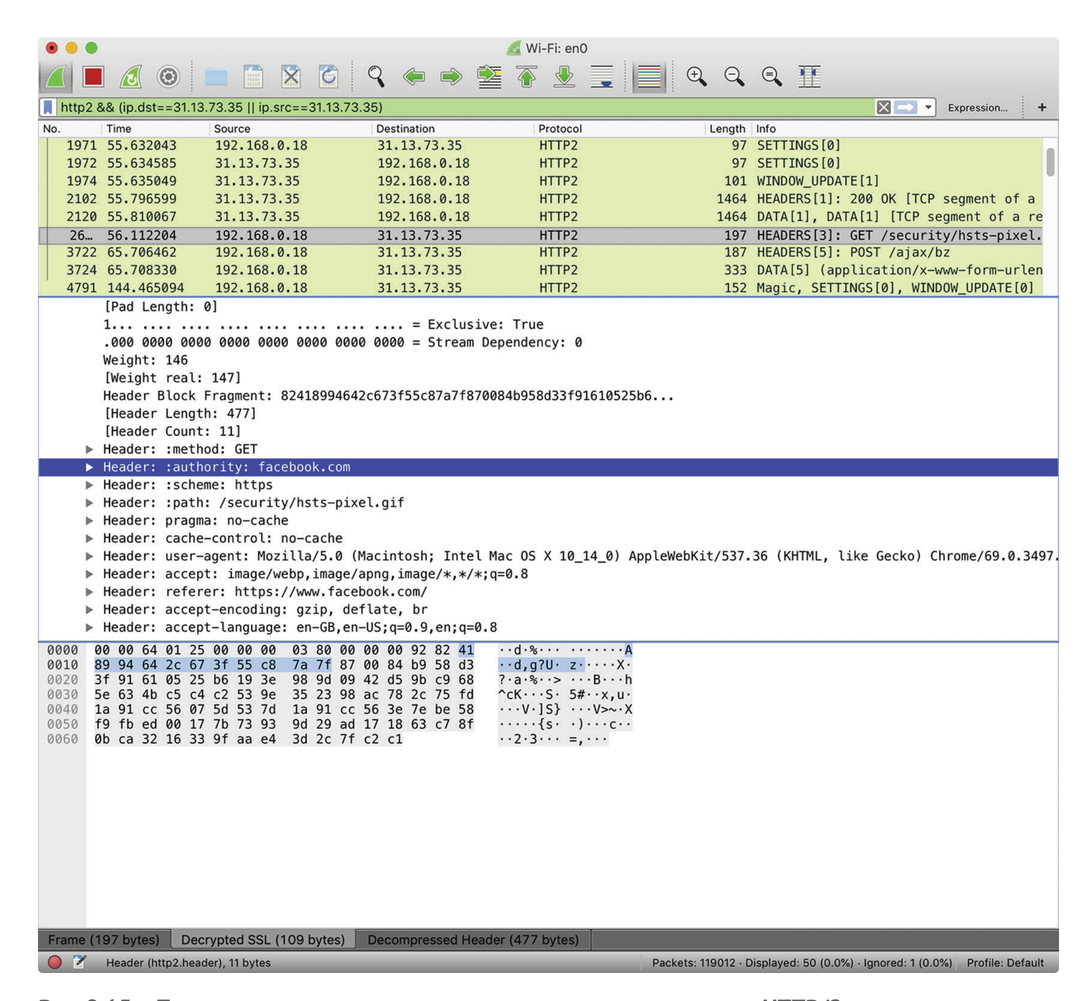


Рис. 8.15 Повторное использование индексированных заголовков HTTP/2 позволяет отправлять большие заголовки.

Заголовок user-agent в первом запросе

```
▶ Header: :method: GET
▶ Header: :authority: www.facebook.com
▶ Header: :scheme: https
▶ Header: :path: /
▶ Header: pragma: no-cache
▶ Header: cache-control: no-cache
▶ Header: upgrade-insecure-requests: 1
▶ Header: user-agent: Mozilla/5.0 (Macintosh; Intel M
▶ Header: accept: text/html,application/xhtml+xml,ap
▶ Header: accept-encoding: gzip, deflate, br
▶ Header: accept-language: en-GB,en-US;q=0.9,en;q=0.
0000  00 01 08 01 25 00 00 00  01 80 00 00 00 ff 82 41
0010  8c f1 e3 c2 f2 8c 85 8c  e7 ea b9 0f 4f 87 84 40
0020  85 ae c1 cd 48 ff 86 a8  eb 10 64 9c bf 58 86 a8
0030  eb 10 64 9c bf 40 92 b6  b9 ac 1c 85 58 d5 20 a4
0040  b6 c2 ad 61 7b 5a 54 25  1f 01 31 7a dc d0 7f 66
0050  a2 81 b0 da e0 53 fa d0  32 1a a4 9d 13 fd a9 92
0060  a4 96 85 34 0c 8a 6a dc  a7 e2 81 04 41 6a 20 7f
0070  6a 43 5d 74 17 91 63 cc  64 b0 db 2e ae cb 8a 7f
0080  59 b1 ef d1 9f e9 4a 0d  d4 aa 62 29 3a 9f fb 52
0090  f4 f6 1e 92 b0 e3 eb 81  76 5a 7d d5 c2 00 29 b8
00a0  72 8e c3 30 db 2e ae cb  9f 53 c0 49 7c a5 89 d3
00b0  4d 1f 43 ae ba 0c 41 a4  c7 a9 8f 33 a6 9a 3f df
00c0  9a 68 fa 1d 75 d0 62 0d  26 3d 4c 79 a6 8f be d0
```

Заголовок user-agent во втором запросе

```
▶ Header: :method: GET
▶ Header: :authority: facebook.com
▶ Header: :scheme: https
▶ Header: :path: /security/hsts-pixel.gif
▶ Header: pragma: no-cache
▶ Header: cache-control: no-cache
▶ Header: user-agent: Mozilla/5.0 (Macintosh; Intel
▶ Header: accept: image/webp,image/apng,image/*,*/*;
▶ Header: referer: https://www.facebook.com/
▶ Header: accept-encoding: gzip, deflate, br
▶ Header: accept-language: en-GB,en-US;q=0.9,en;q=0.
0000  00 00 64 01 25 00 00 00  03 80 00 00 00 92 82 41
0010  89 94 64 2c 67 3f 55 c8  7a 7f 87 00 84 b9 58 d3
0020  3f 91 61 05 25 b6 19 3e  98 9d 09 42 d5 9b c9 68
0030  5e 63 4b c5 c4 c2 53 9e  35 23 98 ac 78 2c 75 fd
```

Рис. 8.16 Заголовок user-agent в первом и втором запросах

Таблица 8.6 Динамический заголовок после получения первого фрейма HEADERS

Индекс	Заголовок	Значение
62	:authority	facebook.com
63	accept-language	en-GB,en-US;q=0.9,en;q=0.8
64	accept-encoding	gzip, deflate, br
65	Accept	text/html,application/xhtml+xml, application/xml;q=0.9,image/webp, image/apng,*/*;q=0.8
66	user-agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36
67	upgrade-insecure-requests	1
68	cache-control	no-cache
69	Pragma	no-cache
70	:authority	www.facebook.com

Такая расшифровка имеет смысл только в том случае, если и отправитель, и получатель синхронизируют свои динамические таблицы HPACK. Также это возможно только в случае сохранения порядка фреймов HEADERS (что осуществляется благодаря гарантированной доставке TCP). Данный процесс может показаться сложным, особенно если вы производите его вручную. К счастью, его можно легко автоматизировать и использовать такие инструменты как, например, Wireshark, что позволит сократить риск возникновения ошибок и облегчит процесс. Однако сейчас вы знаете, как дешифровать заголовки вручную, и этот навык, безусловно, может пригодиться вам в будущем.

Автоматизация процесса расшифровки обеспечивает впечатляющий результат. На рис. 8.15 размер расшифрованного SSL составлял 109 байт, а в первом запросе на рис. 8.13 – 273 байта. Даже при работе с простыми сайтами, загружая три ресурса через одно соединение (остальные загружаются из сегментированного домена) без больших файлов куки или сложных заголовков, вы получаете экономию в размере 68 % байт, которые вы отправляете во фреймах HEADERS, как представлено в табл. 8.7.

Таблица 8.7 Экономия при сжатии HPACK

Запрос	Расшифрованный SSL	Распакованный заголовок	Экономия
1	273	498	45%
2	109	477	77%
3	99	547	82%
Общее	*481*	*1522*	*68%*

Поскольку первый запрос является наименее сжатым, экономия увеличивается по мере использования соединения.

8.6 HPACK в реализациях клиента и сервера

Прежде чем закончить эту главу, рассмотрим еще несколько важных моментов. Во-первых, в статических и динамических таблицах может быть несколько дублированных заголовков, часть которых показана в табл. 8.8.

Таблица 8.8 Примеры повторяющихся заголовков в статических и динамических таблицах HPACK

Индекс	Имя заголовка	Значение заголовка
1	`:authority`	
2	`:method`	`GET`
3	`:method`	`POST`
4	`:path`	`/`
5	`:path`	`/index.html`
...	...	...
19	`accept`	
...	...	...
64	`accept`	`text/html,application/xhtml+xml,application/xml; q=0.9,image/webp, image/apng,*/*;q=0.8`

Ранее я упоминал, что на `:method` можно ссылаться с индексом 2 или 3, и за ним последует значение `DELETE`, поскольку оба индекса относятся к заголовку `:method`. Точно так же `:path` имеет два элемента в таблице для общих значений `:path`. В примере после получения первого запроса то же самое происходит и у `accept`. Спецификация HPACK не дает никаких указаний относительно того, какой индекс следует использовать в подобных случаях. Отправители могут выбрать первый, последний экземпляр или промежуточный (если он существует) вариант; также они могут отказаться от ссылки на определенный ранее номер индекса имени заголовка и добавить еще один (как это сделал Chrome с именем заголовка `:path`).

Еще одним примером того, насколько по-разному браузеры обрабатывают кодирование, является сравнение Chrome и Firefox. В случаях, если в запросе содержится несколько заголовков, они используют разные методы[1]. Например, если вы отправляете два заголовка `cookie`, по-

---

[1] https://stackoverflow.com/questions/49437846/weird-http-2-hpack-encoding-in-firefox.

сле кодировки первого из них в таблице появляется ссылка на его имя: исходная ссылка в статической таблице и еще одна в динамической для значения, которое вы закодировали. Для кодирования второго заголовка файла `cookie` Chrome использует ссылку на статическую таблицу с индексом 32, а Firefox использует заголовок `cookie` из динамической таблицы (в позиции индекса больше 62).

В общем, можно использовать оба метода. Если ссылка в конечном итоге преобразовывается в правильное имя заголовка, отправитель может использовать любой вариант, который ему нравится. Все приведенные мной примеры относятся к повторяющимся названиям заголовков с разными значениями. Однако в спецификации напрямую сказано, что по желанию клиента могут быть продублированы целые пары имя/значение. Клиент может отправить заголовок и значение, которые уже есть в таблице, с инструкциями по их индексированию вместо того, чтобы ссылаться на них из ранее проиндексированных значений[1]:

> *Динамическая таблица может содержать повторяющиеся элементы (элементы с одинаковыми именем и значением). Следовательно, дублирующиеся записи НЕ ДОЛЖНЫ обрабатываться декодером как ошибка.*

Такой процесс приведет к меньшему сжатию этого заголовка при первом использовании дублирующего заголовка, но технически это разрешено.

В конце концов, можно сказать, что отправитель не обязан использовать динамическую таблицу. Например, веб-сервер nginx использует только статическую таблицу[2] предположительно из-за того, что она намного проще в реализации и управлении. Также доступен патч, полностью реализующий кодирование HPACK[3]. По словам его авторов, он улучшает сжатие на 40–95 %, но на момент написания этой книги он еще не включен в базовый код nginx[4].

8.7 Ценность HPACK

На первый взгляд HPACK кажется сложным и пугающим (RFC для него почти равен размеру самой спецификации HTTP/2). В этой главе я вынес наиболее важную информацию о нем. Я надеюсь, что эта глава раскрыла некоторые тайны HPACK, и теперь его техническое описание будет выглядеть не таким пугающим. Большинство пользователей HTTP/2 и веб-разработчиков может проигнорировать сложные детали и просто согласиться с тем, что заголовки HTTP/2 сжимаются довольно эффективно,

---

1 https://httpwg.org/specs/rfc7541.html#dynamic.table.

2 https://trac.nginx.org/nginx/changeset/12cadc4669a7/nginx.

3 https://github.com/cloudflare/sslconfig/blob/master/patches/nginx_1.13.1_http2_hpack.patch.

4 https://twitter.com/igrigorik/status/1029827634815856640.

что приводит к значительной экономии места, особенно на стороне запроса, где заголовки составляют большую часть данных. Cloudflare, одна из крупнейших сетей CDN, с помощью HPACK сократила объем данных запросов на 53 %[1]. Что касается ответов, то, несмотря на то что HTTP-заголовки могут быть большими, они обычно уступают в размере телам HTTP, поэтому здесь экономия кажется менее значительной (Cloudflare в среднем получил лишь 1,4 % экономии). Но большинство потребительских сетевых подключений имеет ограниченную полосу пропускания для исходящего трафика по сравнению со скачиванием, и запрос ресурсов является первым этапом, поэтому, возможно, на стороне запроса выгода в любом случае более важна.

Резюме

- Существуют разные методы сжатия данных.
- Заголовки HTTP содержат конфиденциальные данные, такие как файлы куки, поэтому они не могут сжиматься по тому же алгоритму, что и тела HTTP; это может привести к утечке данных в результате различных атак.
- HPACK – это формат сжатия, созданный специально для сжатия HTTP-заголовков в HTTP/2.
- Формат HPACK имеет особый двоичный формат, в котором используется заранее создаваемая статическая таблица распространенных имен заголовков (и в некоторых случаях значений) и динамическая таблица, создаваемая во время сеанса.
- Значения, не являющиеся ссылками на таблицы, могут быть представлены в формате ASCII или кодировки Хаффмана.
- При использовании кодировки Хаффмана значения получаются меньше.
- В HPACK существует несколько способов отправки HTTP-заголовков, и браузеры могут кодировать их по-разному.

[1] https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/.

Часть IV

Будущее HTTP

Если вы дочитали книгу до этой части, вероятнее всего, вы уже обладаете углубленными знаниями о протоколе HTTP/2 и можете полностью понять его спецификацию. Данная часть – заключительная, и здесь я рассмотрю будущее HTTP. Сегодня у нас есть HTTP/2, на основе которого работает большее количество веб-сайтов, однако его создатели не почивают на лаврах. В некотором смысле HTTP/2 – уже не новинка, и люди уже задумываются о создании следующих улучшенных версий.

В главе 9 я расскажу о QUIC – экспериментальном протоколе, который вполне может стать преемником HTTP/2 и решить проблемы более низкого уровня TCP. Сегодня QUIC прежде всего нуждается в стандарте (и, возможно, к моменту публикации этой книги, его уже разработают), однако немаловажным аспектом остается и повсеместное распространение этого протокола среди пользователей, что, по моему мнению, произойдет несколько позже. QUIC во многом схож с HTTP/2, поэтому читатели, дошедшие до этой части, имеют преимущество, которое облегчит им изучение протокола, а может быть, даже поможет принять участие в его внедрении.

В главе 10 я рассказываю о возможных перспективах развития HTTP. Изначально он был надежным и открытым для различного рода расширений, HTTP/2 как никто лучше справился с развитием этой концепции, и поэтому мы можем сделать вывод о том, что существует масса возможных вариантов его дальнейшего развития.

9 TCP, QUIC и HTTP/3

В этой главе мы рассмотрим:

- слабые стороны TCP;
- оптимизацию TCP;
- основы QUIC;
- различия между QUIC и HTTP/2.

HTTP/2 был нацелен на устранение недостатков протокола HTTP. По большей части проблемы решены введением концепции использования одного мультиплексированного соединения. В HTTP/1.1 использование соединений было нерациональным, поскольку они создавались для каждого ресурса и носили «одноразовый» характер. Если при ответе на запрос возникали задержки (например, из-за того, что сервер был занят созданием запрошенного ресурса), соединение блокировалось. HTTP/2 позволяет использовать одно соединение сразу для нескольких ресурсов, поэтому в таких случаях оно остается доступным для каждого из них.

Помимо рационального использования соединений, HTTP/2 позволил повысить показатели производительности, поскольку сами по себе HTTP-соединения малоэффективны. Создание HTTP-соединения требует затрат; в противном случае мультиплексирование не несет пользы. Затраты связаны не с самим протоколом, а с двумя его базовыми технологиями, посредством которых и создается соединение: TCP и TLS (последний предназначен для обеспечения безопасности).

В этой главе я проанализирую недостатки упомянутых технологий и покажу, что, несмотря на то что HTTP/2 справляется с большинством из

них, при некоторых сценариях по их же причине он может быть медленнее, чем HTTP/1.1. Затем я расскажу о QUIC, который позволяет решить эту проблему.

9.1 HTTP и слабые стороны TCP

Эффективная работа HTTP напрямую зависит от надежности сетевого соединения, которое должно гарантировать надежную и упорядоченную доставку данных. До недавнего времени такие соединения устанавливались с помощью протокола TCP (Transmission Control Protocol). С его помощью происходит коммуникация между двумя конечными точками (обычно это браузер и веб-сервер). TCP заботится о передаче сообщений, обеспечивая их прибытие, обрабатывает повторные передачи, если сообщения не поступают, а также следит за порядком их отправки на любом уровне. HTTP не требует реализации каких-либо из этих критериев; он лишь предполагает, что все они были соблюдены. И именно на этом предположении и построен весь протокол.

TCP обеспечивает сохранность данных путем присвоения каждому TCP-пакету порядкового номера; в случае, если они доставлены не по порядку, по прибытии TCP упорядочивает их согласно номерам и при необходимости запрашивает недостающие. TCP работает на базе CWND (прародителя управления потоками в HTTP/2; см. главу 7). Размер CWND соответствует максимальному объему отправленных данных. Данное окно уменьшается по мере отправки сообщений и увеличивается снова при подтверждении пакетов. Его начальный размер невелик, но со временем, если пропускная способность сети выдерживает подаваемые нагрузки, оно становится все больше. Если клиент не справляется, окно уменьшается. Данный процесс был настолько отлажен, что TCP/IP стал основой Internet. Однако основной принцип работы TCP порождает пять проблем, касающихся HTTP:

- начальную задержку. Порядковые номера, использующиеся и отправителем, и получателем, должны согласовываться в начале соединения;
- ограничение производительности *алгоритмом замедленного старта TCP*. Во избежание повторных передач он строго контролирует объем отправляемых данных;
- снижение скорости TCP в случаях неполного использования соединения. Если соединение используется не полностью, TCP уменьшает размер CWND, поскольку нельзя быть уверенным, что параметры сети не изменились с момента установки последнего оптимального размера CWND;
- снижение скорости из-за потери пакетов. TCP предполагает, что вся потеря пакетов происходит из-за перегрузки, что далеко не всегда так;
- образование очередей пакетов. С целью соблюдения строгой упорядоченности пакеты, полученные раньше (или позже), чем нужно, задерживаются и ставятся в очередь.

В HTTP/2 все эти проблемы сохранились, и некоторые из них выступают причинами, по которым иногда предпочтительнее использовать одно TCP-соединение. По причине двух последних из них HTTP/2 может работать даже медленнее, чем HTTP/1.1; при этом в HTTP/2 возрастает вероятность потерь.

9.1.1 Задержка предустановки HTTP/2

В главе 2 я рассказывал о трехстороннем TCP-рукопожатии. Оно в сочетании с настройкой HTTPS, которая зачастую является обязательным требованием HTTP (и всех браузеров, работающих на HTTP/2), приводит к значительной задержке перед отправкой первого HTTP-сообщения, как показано на рис. 9.1.

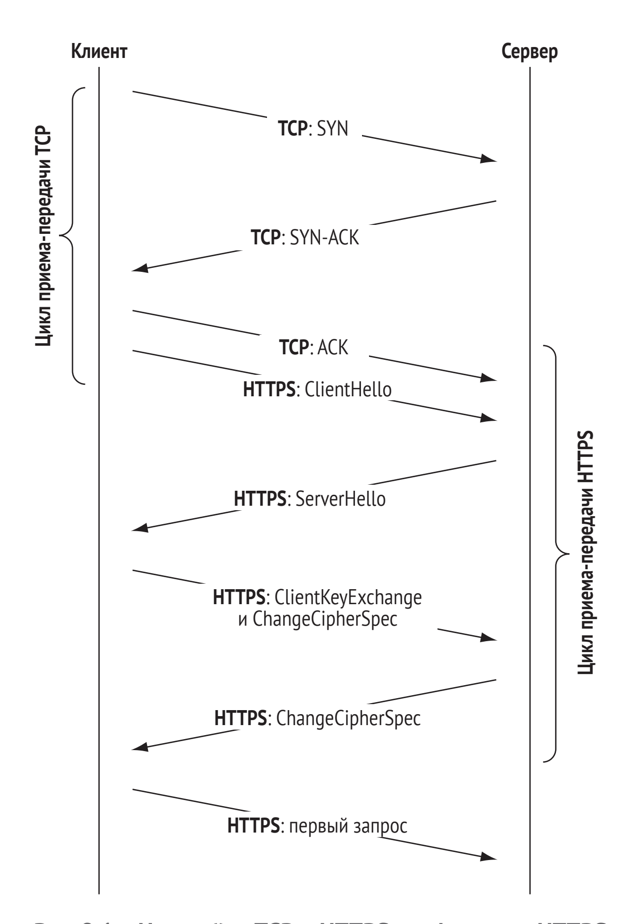


Рис. 9.1 Настройка TCP и HTTPS-трафика для HTTPS-соединения

В зависимости от размера сообщений HTTPS-рукопожатия, потребуется не менее трех циклов приема-передачи (1,5 для TCP, 2 для HTTPS с перекрытием 0,5). После этого вы установите соединение с сервером

и сможете отправить первый запрос. Кроме того, на рис. 9.1 не представлен процесс поиска DNS, который обычно приводит к дополнительным задержкам.

Процесс прохождения всех этих шагов в реальной жизни подразумевает возникновение значительных задержек; особенно критично это было для HTTP/1.1, хотя HTTP/2 также подвержен их возникновению. На рис. 9.2 представлена каскадная диаграмма для веб-сайта Amazon из главы 2, на которой выделены все задержки соединения.

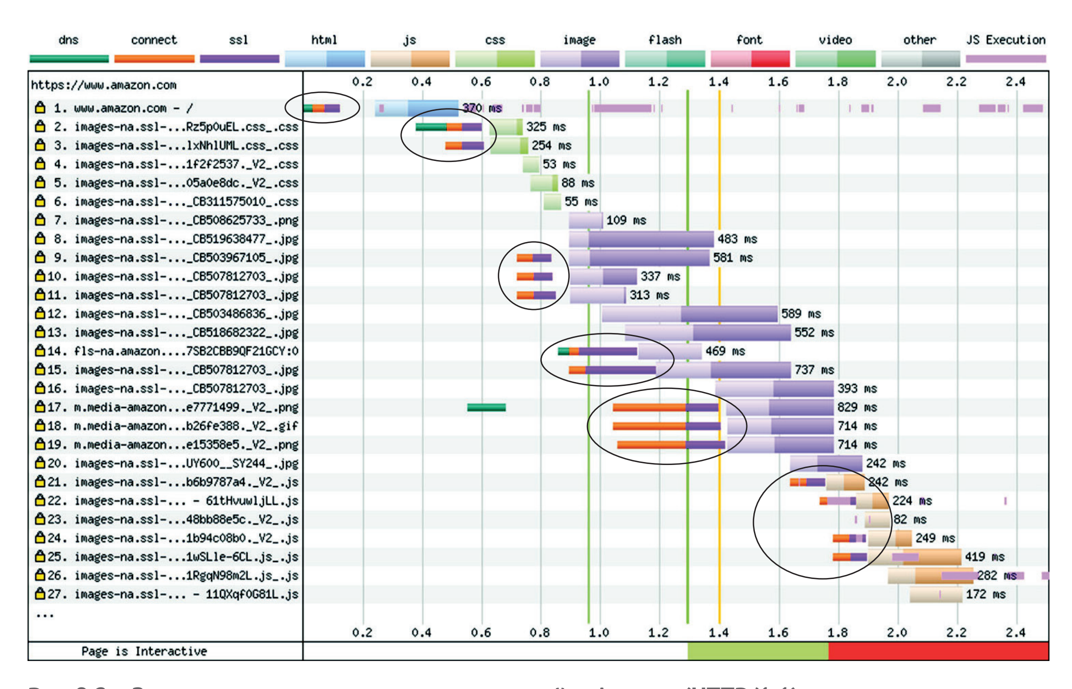


Рис. 9.2 Задержки начального соединения на сайте Amazon (HTTP/1.1)

Конечно, для HTTP/2 предпочтительнее использовать одно соединение, но это не отменяет факта возникновения задержки начального соединения. Кроме того, задержкам подвержены и отдельные домены, которые по какой-либо причине не могут быть *объединены* (см. главу 6). Даже после перехода на HTTP/2 Amazon не полностью избавился от проблемы задержки предустановки. Кроме того, на каскадной диаграмме мы можем увидеть задержки не только для начального, но и для всех последующих соединений, которые не могут использовать упомянутое выше соединение HTTP/2 (потому что его не принимает сервер или оно не аутентифицировано), как показано на рис. 9.3.

Как видите, HTTP/2 сокращает как количество соединений, так и количество задержек. На рис. 9.3 видно, что было устранено около 15 задержек, однако было бы намного лучше устранить их все.

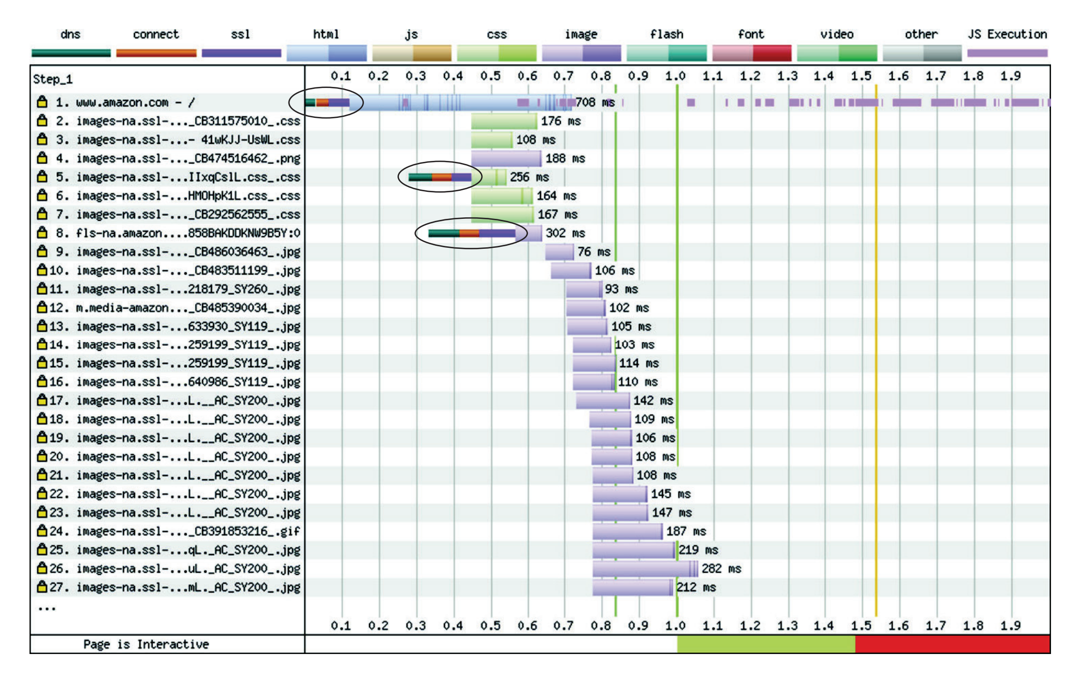


Рис. 9.3 **Начальные задержки значительно сократились, но не были устранены полностью (HTTP/2)**

9.1.2 Неэффективность системы контроля перегрузки в TCP

Даже при успешной установке соединения недостатки TCP могут привести к другим проблемам с производительностью. В первую очередь так происходит ввиду строгого характера TCP: все TCP-пакеты приходят исключительно в установленном порядке. Ввиду этого требования в протокол включены такие аспекты, как, например, *контроль перегрузки*.

Контроль перегрузки призван предотвратить *крах сети*, который возникает, когда сеть тратит больше времени на повторную передачу отброшенных пакетов, чем на отправку новых. В 1980-е годы, в эпоху расцвета Internet, эта проблема была близка к воплощению в реальность[1]. Именно тогда в TCP добавили различные функции контроля перегрузки, которые развиваются и по сей день. Алгоритмы контроля привносили некоторую стабильность, но одновременно с этим имели и ряд минусов, особенно для HTTP.

Алгоритм замедленного старта TCP

Алгоритм замедленного старта позволяет определить оптимальную пропускную способность. Начиная работу на низкой скорости, он посте-

---

[1] https://tools.ietf.org/html/rfc2914#section-3.1.

пенно повышает ее и в течение этого времени тщательно отслеживает соединение и его пропускную способность.

Объем данных, которые могут быть отправлены по TCP-соединению, определяется размером окна перегрузки. Его минимальный размер для современных ПК и серверов – 10 сегментов, каждый из которых равняется 1460 байтам (MSS), или около 14 Кб; данное изменение появилось относительно недавно[1], поскольку многие серверы все еще ограничиваются четырьмя сегментами, как это было принято ранее). В процессе замедленного старта с каждым циклом приема-передачи размер окна перегрузки увеличивается вдвое, как представлено в табл. 9.1.

Таблица 9.1 Увеличение размера окна перегрузки при использовании замедленного старта

Цикл приема-передачи	MSS	Размер CWND (количество сегментов)	Размер CWND (в Кб)
1	1460	10	14
2	1460	20	28
3	1460	40	57
4	1460	80	114
5	1460	160	228

Увеличение размера вдвое приводит к стремительному (экспоненциальному) росту пропускной способности, и после нескольких циклов она достигает оптимального установленного получателем значения (см. первую часть рис. 9.4).

Зачастую предел оптимального значения намного ниже, чем на рис. 9.4, и обычно составляет 65 Кб. Именно это значение было изначальным пределом для TCP (см. раздел 9.1.4). По достижении максимального показателя, полагая, что пакеты не были потеряны, TCP входит в фазу предотвращения перегрузки, и размер окна продолжает увеличиваться, однако уже с гораздо меньшей скоростью (рост линейный). Рост продолжается до тех пор, пока протокол не начнет замечать отброшенные пакеты и, как предполагается, не достигнет оптимальной пропускной способности, как показано во второй части рис. 9.4.

К сожалению, при использовании HTTP на начальном этапе загрузки страницы, вероятнее всего, потребуется именно максимальная пропускная способность. Например, в сеансе Facebook размер одной только домашней страницы составляет 125 Кб, однако такого значения пропускная способность TCP не достигает вплоть до четвертого цикла приема-передачи. После начальной загрузки веб-страницы и ее файлов потребность в загрузке большого количеств файлов снижается, и зачастую этот момент совпадает с достижением TCP наибольшей пропускной способности.

1 https://tools.ietf.org/html/rfc6928.

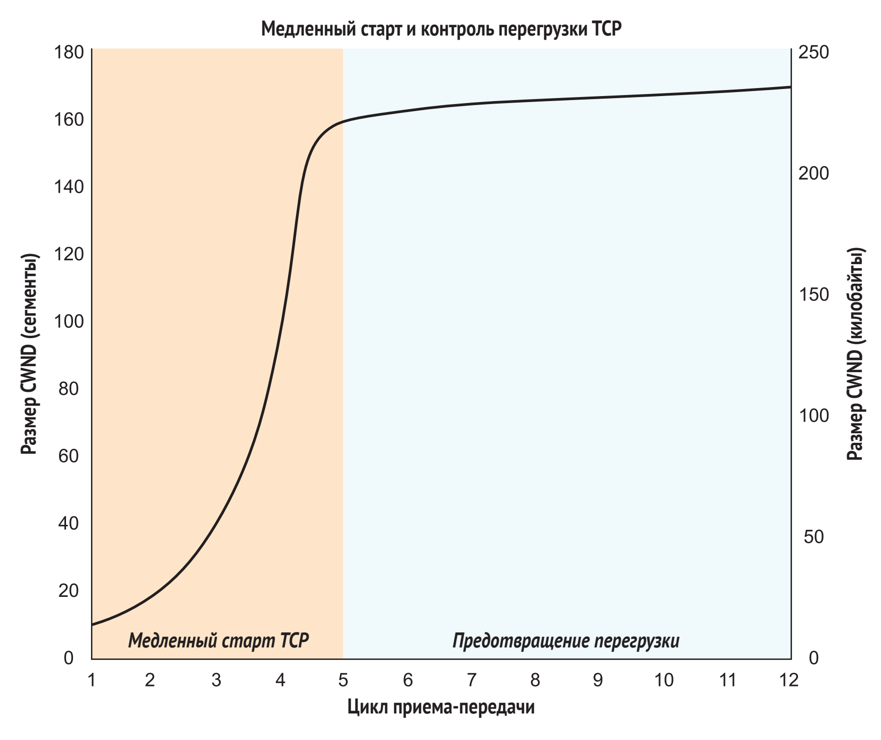


Рис. 9.4 Увеличение пропускной способности до оптимального значения при замедленном старте TCP

Действительно ли замедленный старт – замедленный?

Ввиду экспоненциального характера замедленного старта, он, казалось бы, по определению не может быть медленным. Фактически фаза предотвращения перегрузки занимает намного больше времени. Название «замедленный старт» относит нас к небольшому размеру окна перегрузки на начальном этапе, поэтому его и называют замедленным стартом, а не замедленным ростом.

Он определенно медленнее, чем запуск соединения с большим значением окна, заданным сервером, и последующее его уменьшение. Каждое TCP-соединение проходит через этот этап, поэтому сам протокол изначально медленный (и это сделано намеренно).

Втисните как можно больше в первые 14 Кб

В интернет-публикациях очень часто встречается следующий совет по повышению производительности – вам рекомендуют вместить все критические ресурсы в первые 14 Кб HTML-кода. В теории эти 14 Кб должны передаваться в первых 10 TCP-пакетах, что поможет предотвратить задержки при установке TCP-соединения. Например, в начальные 14 Кб должен быть включен любой критически важный встроенный CSS (при условии, что браузер готов

начать обработку неполных HTML-страниц; к слову, многие браузеры поддерживают эту функцию).

При установке HTTPS-соединения, и в частности HTTP/2-соединения, ситуация меняется. Некоторые из первых 10 TCP-пакетов будут использоваться по крайней мере для:

- двух HTTPS-ответов (`Server Hello` и `Change Spec`);
- двух HTTP/2-фреймов `SETTINGS` (один из них отправляет сервер, а также он подтверждает аналогичный фрейм клиента);
- одного фрейма `HEADERS` в ответ на первый запрос.

Таким образом, остается в лучшем случае 5 пакетов (около 7 Кб). На самом деле, любое из приведенных выше сообщений может требовать более одного TCP-пакета. Кроме того, свою долю возьмут и фреймы `WINDOW_UPDATE` или `PUSH_PROMISE`.

К счастью, подтверждает отправку этих пакетов клиент, что значительно увеличивает размер CWND. В некотором смысле задержки, возникающие в начале соединения из-за HTTPS, могут означать, что размер CWND уже и так увеличен. Однако этот факт компенсируется затратами на задержку предустановки самого HTTPS.

Таким образом, идея размещения критически важных ресурсов в начало обмена данными HTML имеет право на жизнь. Хотя, на мой взгляд, при использовании HTTPS или HTTP/2 не стоит зацикливаться именно на 14 Кб.

В главе 2 я говорил, что HTTP/2 с его концепцией использования одного соединения имеет преимущество перед HTTP/1.1, однако если углубляться в подробности, то это утверждение не совсем точно. С одной стороны, использование HTTP/1.1 более затратное, поскольку предполагает установку сразу нескольких соединений (обычно шесть на домен и больше при использовании сегментирования). В сравнении с HTTP/2, на начальном этапе HTTP/1.1 получает больше начальных CWND (при условии, что все соединения открываются одновременно). Взгляните на табл. 9.2.

Таблица 9.2 Рост скорости при замедленном старте TCP с шестью соединениями

Цикл приема-передачи	MSS	Размер CWND (количество сегментов)	Размер CWND (в Кб)	Размер CWND для шести подключений (Кб)
1	1460	10	14	85
2	1460	20	28	171
3	1460	40	57	342
4	1460	80	114	684
5	1460	160	228	1368

С другой стороны, если изначально используется только одно соединение (как это зачастую бывает при загрузке веб-страницы HTML), любые дополнительные новые соединения по HTTP/1.1 устанавливаются с более медленного, нижнего предела, чем соединение HTTP/2, которое

к этому времени уже, вероятнее всего, достигает использования максимальной пропускной способности. Таким образом, замедленный старт TCP в некоторой степени значим для обеих версий протокола.

Простаивающие соединения снижают производительность

Алгоритм замедленного старта TCP вызывает задержки при установке соединения, а также когда соединение простаивает. В период бездействия состояние сети может измениться, поэтому TCP уменьшает размер CWND и перезапускает механизм замедленного старта с целью поиска оптимального размера.

К сожалению, во время просмотра веб-страниц потреблению трафика свойственны всплески (например, при переходе на новую страницу), а при чтении пользователем страницы соединение бездействует. Такая смена состояний циклична, поэтому возврат к начальному состоянию во время периодов простоя может влиять на просмотр веб-страниц не самым лучшим образом.

Например, веб-сайт Amazon при использовании HTTP/1.1 загружает страницы из основного домена, а большую часть ресурсов из поддоменов. Таким образом, первое установленное соединение простаивает, как показано на рис. 9.5.

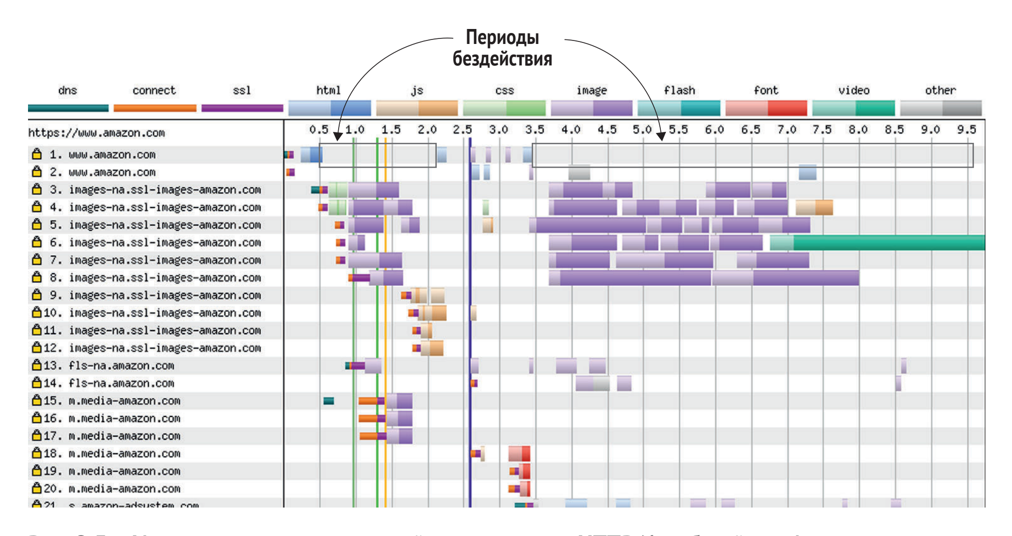


Рис. 9.5 Использование соединений по протоколу HTTP/1 веб-сайтом Amazon

Простой отрицательно влияет как на самое первое установленное соединение (поскольку оно, вероятно, будет использоваться снова при последующей навигации по страницам), так и на последующие. На рис. 9.5 видно, что многие соединения не так активны. Пробелы на диаграмме говорят о том, что соединения используются неэффективно (о чем я рассказывал в главе 2). На TCP простои влияют наиболее отрицательно, ведь в это время он полностью ограничивает неактивное соединение. Когда оно снова понадобится (например, при навигации по страницам),

процесс «разгона» начинается почти с самого начала. Если соединение остается открытым, не нуждаются в повторной настройке лишь TCP-рукопожатие и HTTPS.

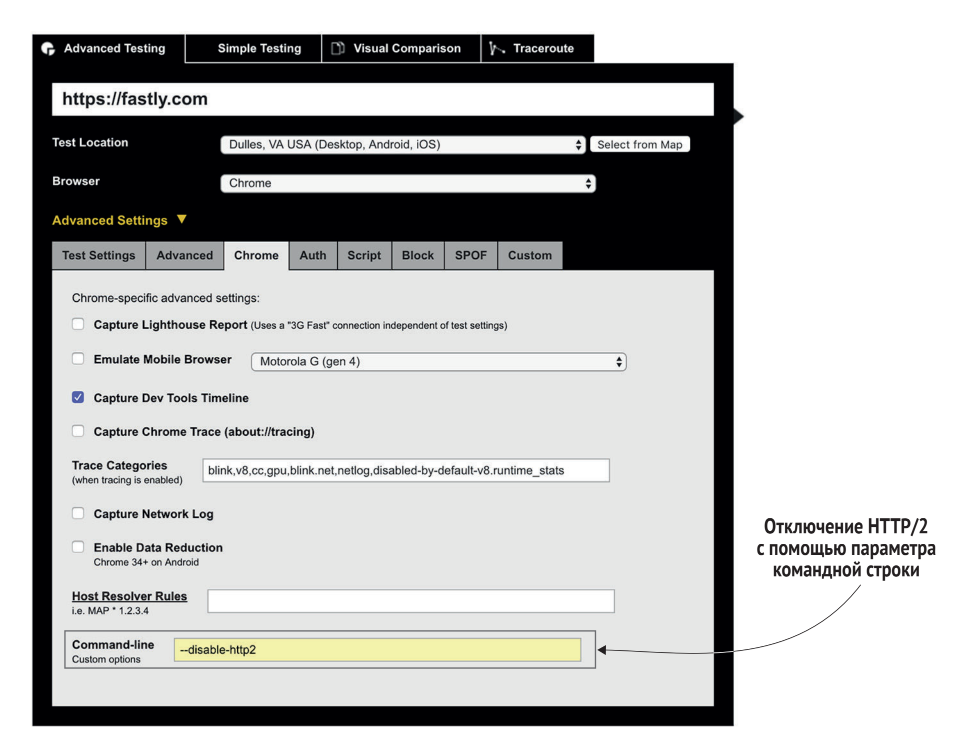


Рис. 9.11 Отключение HTTP/2 в Chrome

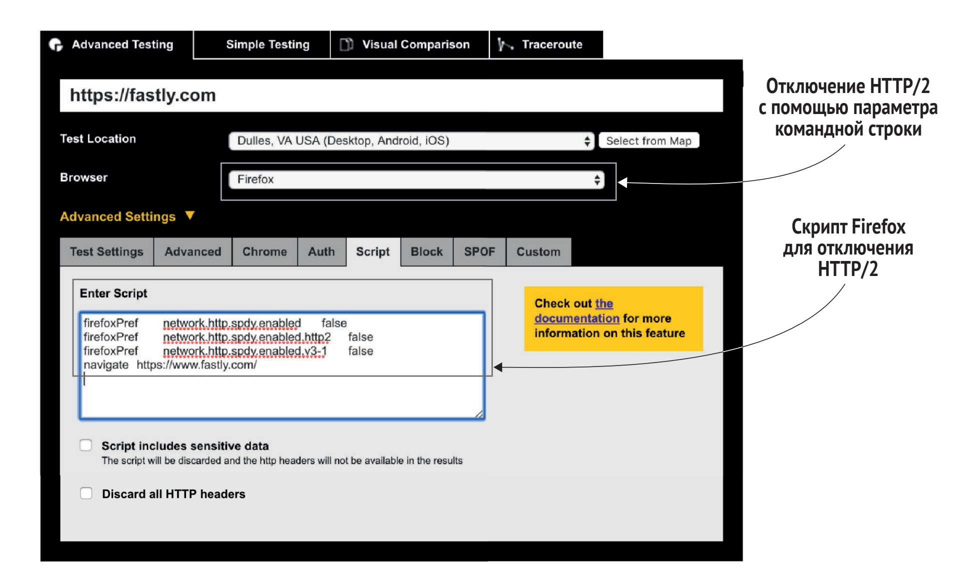


Рис. 9.12 Отключение HTTP/2 в Firefox для WebPagetest

HTTP/2 с 2 % потерей пакетов

Результаты производительности (средние значения)

	Load Time	First Byte	Start Render	Speed Index	First Interactive (beta)	Document Complete			Fully Loaded			
						Time	Requests	Bytes In	Time	Requests	Bytes In	Cost
First View (Run 7)	3.101s	0.299s	0.800s	3.386s	> 5.485s	3.101s	38	843 KB	7.310s	135	2,265 KB	$$$$$

HTTP/1.1 с 2 % потерей пакетов

Результаты производительности (средние значения)

	Load Time	First Byte	Start Render	Speed Index	First Interactive (beta)	Document Complete			Fully Loaded			
						Time	Requests	Bytes In	Time	Requests	Bytes In	Cost
First View (Run 4)	2.672s	0.374s	1.000s	4.790s	6.637s	2.672s	38	826 KB	11.998s	137	2,487 KB	$$$$$

Рис. 9.13 Загрузка домашней страницы Ebay по HTTP/2 и HTTP/1.1 с потерей 2 % пакетов

Как видите, HTTP/2 почти на полсекунды медленнее. Повторение того же теста без потери пакетов показывает, что HTTP/2 работает, как и ожидается, быстрее.

Также, кликнув на Raw Page Data в правой части страницы, можно экспортировать необработанные данные, как показано на рис. 9.14.

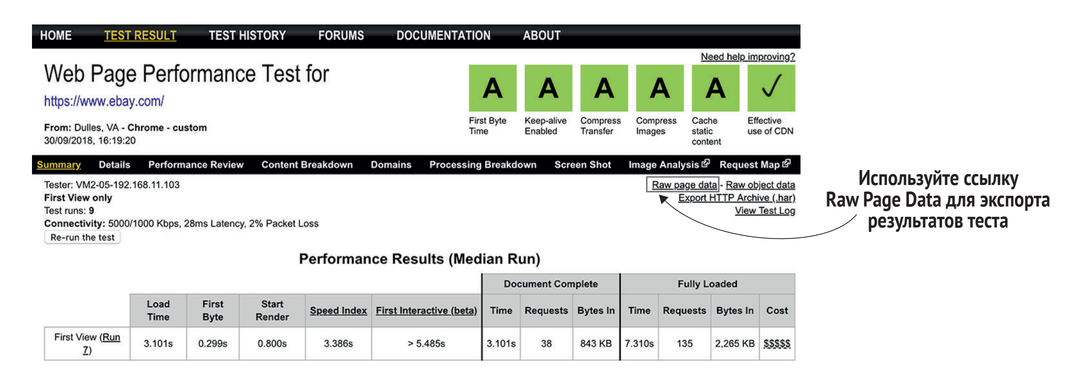


Рис. 9.14 Экспорт необработанных данных WebPagetest в файл CSV

Функция экспорта может быть полезна при выполнении нескольких тестов и последующем отображении результатов на графике. Кроме того, кликнув на Test History и выбрав оба изображения, вы увидите их сравнение, как показано на рис. 9.15.

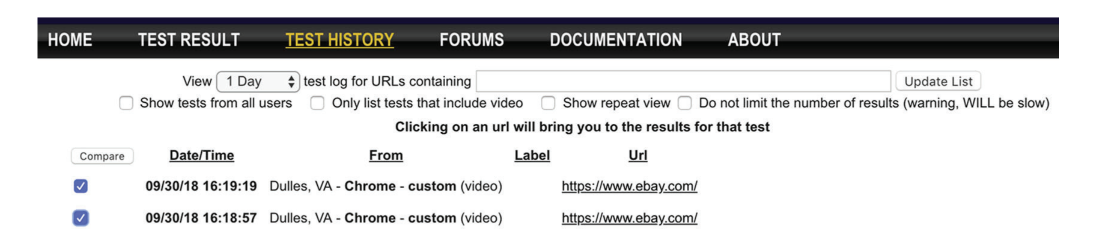


Рис. 9.15 Выбор двух результатов для сравнения

Такая функция дает вам доступ к множеству данных, таких как размер относительно шкалы времени и эскизы страницы, как показано на рис. 9.16.

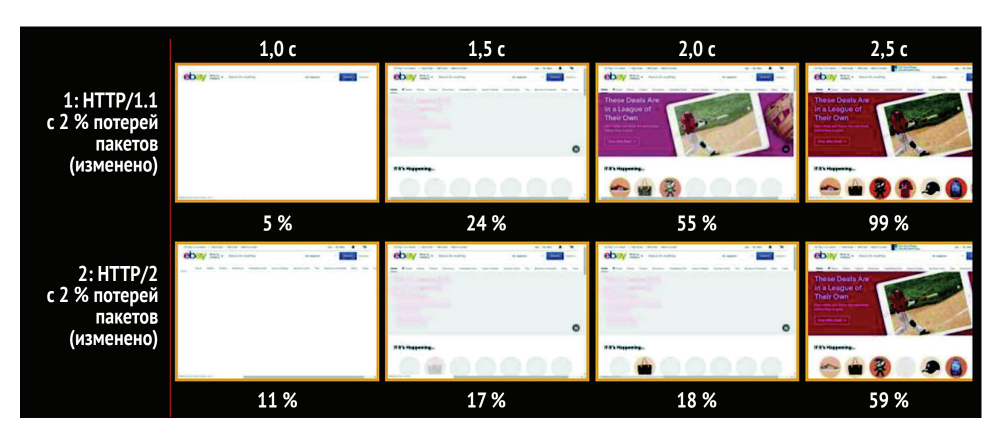


Рис. 9.16 Сравнение двух результатов тестов на WebPagetest

WebPagetest – это прекрасный инструмент для проведения подобных сравнений производительности. Также вы можете хранить свои данные, если планируете проводить большое количество проверок или хотите протестировать серверы разработки, недоступные для веб-версии.

Стоит ли воздерживаться от перехода на HTTP/2 из-за недостатков TCP?

Стоит ли отказываться от перехода на HTTP/2 ввиду того, что при потере пакетов производительность может значительно снизиться? Разумеется, нет. Помните, что в большинстве сценариев HTTP/2 работает намного быстрее HTTP/1.1. Стоит ли лишать пользователей комфортной работы с вашим веб-сайтом только из-за редких случаев неэффективности?

В этой главе говорится о том, что такие проблемы могут возникать на практике, однако вероятность их возникновения может быть разной. Лучше всего опираться на результаты исследования собственных сайтов, а не на искусственные сценарии, подобные тем, что описаны в этой главе. Как я уже говорил ранее, реальная сеть, в которой наблюдается постоянная потеря 2 % пакетов, скорее всего, работает плохо во всех отношениях. К сожалению, в реальной жизни измерить потерю пакетов и составить статистику намного сложнее. Однако научные исследования, основанные на более реалистичных сценариях потери пакетов[a], показали, что, как и ожидалось, в целом HTTP/2 превосходит HTTP/1.1.

Реализации HTTP/2 все еще относительно молоды, и со временем в них будут вноситься улучшения. То же самое и с веб-сайтами, которые, скорее всего, будут оптимизироваться для HTTP/2. В конечном итоге даже сам TCP еще совершенствуется и в дальнейшем может быть оптимизирован (см. раз-

дел 9.1.4). Иногда предлагают использовать для HTTP/2 несколько TCP-соединений, полагая, что это позволит решить существующие проблемы. Однако такой вариант сводит на нет все причины перехода на HTTP/2. Данная версия протокола работает лучше именно по причине использования одного соединения.

При определенных сценариях, когда, например, посетители вашего сайта пользуются слабыми сетевыми соединениями, не поддающимися оптимизации, может быть разумным остаться на HTTP/1.1 или прибегнуть к сегментированию HTTP/2-соединений. В конце концов, лучшим вариантом в любом случае будут являться измерение и проверка любых изменений.

[a] https://www.semanticscholar.org/paper/HTTP%2F2-Performance-in-Cellular-Networks-Goel-Steiner/63fa6b3310a7c4d799d5b0b5bf37f0620dd3fc5d?tab=abstract.

9.1.4 Оптимизация TCP

Как выяснилось, TCP может иметь значительное влияние на производительность HTTP. HTTP/2 решил большую часть проблем протокола HTTP, однако недостатки на других уровнях стали более заметными. Благодаря *мультиплексированию* в HTTP/2 проблема блокировки HOL на уровне HTTP отошла на второй план, однако более явной стала проблема блокировки HOL на уровне TCP, особенно в средах, которым присуща потеря пакетов.

Существует только два пути решения этой проблемы: улучшишь TCP или же отказаться от него. Далее в этом разделе мы рассмотрим способы реализации первого пути, а в разделе 9.2 – второго. За прошедшие несколько лет TCP уже претерпел несколько улучшений; некоторые из них, вероятнее всего, уже используются в определенных средах.

Обновление операционной системы

Лучший способ оптимизировать TCP – обновить вашу операционную систему. Несмотря на то что протокол TCP был создан в 1974 году и вообще является довольно старым, его все еще исследуют и улучшают. К сожалению, обычно этот протокол работает под управлением операционной системы компьютера, и возможностей улучшить его другими способами не так уж много. Именно поэтому основным критерием оптимальной работы TCP является актуальность версии операционной системы.

Здесь я буду рассматривать примеры на Linux, однако все методы в равной степени применимы и к другим операционным системам, таким как Windows и macOS; несмотря на то что настройка в них происходит несколько иначе, осуществить ее все же достаточно просто. Также при необходимости я указываю версию Linux, в которой было реализовано то или иное улучшение. В табл. 9.4 представлены версии ядра Linux в самых популярных дистрибутивах.

Таблица 9.4. Версии ядра Linux в популярных дистрибутивах

Комплект поставки	Версия ядра Linux
RHEL/Centos 6	2.6.32
RHEL/Centos 7	3.10.0
Ubuntu/Debian 14.04	3.13
Ubuntu/Debian 16.04	4.4
Ubuntu/Debian 18.04	4.15
Debian 8 Jessie	3.16
Debian 9 Stretch	4.9

Поиск и изменение настроек TCP-соединения

Большинство настроек TCP в Linux доступно для просмотра в следующей директории:

```
/proc/sys/net/ipv4/
```

Несмотря на ее название, большинство настроек здесь применимо и к TCP-соединениям IPv6. Вы можете просмотреть значения с помощью `cat`:

```
$ cat /proc/sys/net/ipv4/tcp_slow_start_after_idle
1
```

Установить значения вы можете с помощью команды `sysctl`:

```
sysctl -w net.ipv4.tcp_slow_start_after_idle=0
```

Будьте осторожны при изменении любого из параметров, поскольку TCP – это очень важная часть системы. Я советую по возможности вообще не менять их. Вместо этого благодаря полученным знаниям вы сможете проверить, насколько вам подходят уже имеющиеся настройки, и в дальнейшем учитывать эту информацию при принятии решения о необходимости обновления операционной системы.

Увеличение начального размера CWND

При замедленном старте TCP для увеличения размера CWND понадобится полный цикл приема-передачи. Изначально стартовый размер соответствовал размеру 1 TCP-пакета, затем увеличился до 2, а позже до 4; в ядре Linux 2.6.39 значение по умолчанию увеличилось с 4 до 10. Данный параметр обычно жестко запрограммирован в коде ядра, поэтому менять его не рекомендуется никаким способом, кроме как путем обновления операционной системы.

Масштабирование окна

Обычно наибольший допустимый размер окна CWND составлял 65 535 байт, однако новые версии позволили применять к этому значению коэффициент масштабирования, расширяя возможный размер до 1 Гб. Данный параметр был установлен по умолчанию в ядре Linux 2.6.8, по-

этому, скорее всего, большинству читателей эта функция доступна. Однако для пущей уверенности вы можете проверить это следующим образом:

```
$ cat /proc/sys/net/ipv4/tcp_window_scaling
1
```

Использование выборочного подтверждения

Выборочное подтверждение (Selective Acknowledgement, SACK) позволяет TCP подтверждать получение пакетов в произвольном порядке, что помогает избежать повторной отправки потерянных пакетов. Если пакеты 1–10 уже отправлены, но пакет 4 был утерян, вы можете подтвердить полученные пакеты 1–3 и 5–10. Таким образом, повторно должен быть отправлен только пакет 4. При отсутствии этой функции пришлось бы отправлять повторно пакеты 4–10. Вы можете включить выборочное подтверждение с помощью следующей команды:

```
$ cat /proc/sys/net/ipv4/tcp_sack
1
```

Отключение повторного запуска алгоритма замедленного старта

Возможно, вам захочется отключить повторный запуск алгоритма замедленного старта. После периода простоя TCP-соединение восстанавливается с расчетом на изменение условий сети, поэтому значения, которые были актуальны до этого, могут стать недействительными. Всем веб-серверам присущи скачки, когда пользователи просматривают сайт, и падения трафика, когда пользователи читают веб-страницы или переходят между ними. Таким образом, не всегда требуется повторный запуск замедленного старта.

Данный параметр обычно включен по умолчанию:

```
$ cat /proc/sys/net/ipv4/tcp_slow_start_after_idle
1
```

Чтобы отключить его, используйте следующую команду:

```
sysctl -w net.ipv4.tcp_slow_start_after_idle=0
```

Как я и говорил, от изменения настроек TCP в системе лучше воздержаться. Однако в зависимости от предназначения вашего сервера (например, он является специализированным веб-сервером), возможно, стоит подумать об изменении параметра повторного запуска замедленного старта.

Использование TCP Fast Open

TCP Fast Open в самом начале позволяет отправить пакет трафика с начальной `SYN`-частью трехстороннего TCP-рукопожатия. Данный метод исключает задержку установки соединения, связанную с TCP (см. раздел 9.1.1). По соображениям безопасности этот пакет может быть от-

правлен только в повторном TCP-соединении. Кроме того, метод требует поддержки как со стороны клиента, так и со стороны сервера. TCP Fast Open позволяет ускорить процесс рукопожатия и приблизить момент отправки первого HTTP- или HTTPS-сообщения, как показано на рис. 9.17.

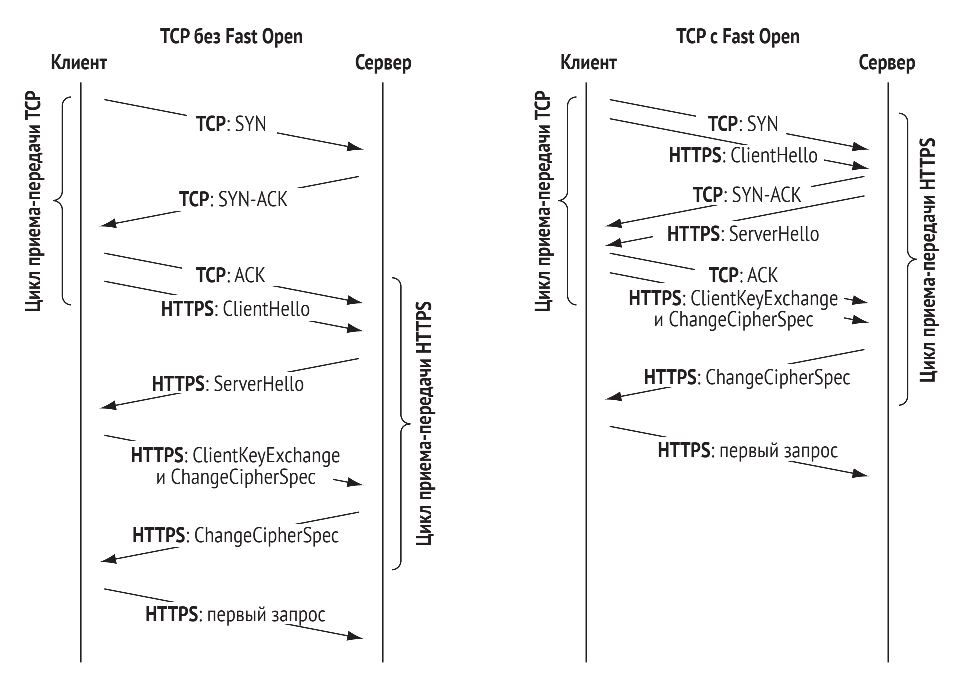


Рис. 9.17 Рукопожатие повторного TCP- и HTTPS-соединения с функцией Fast Open и без нее

В Linux вы можете проверить поддержку этой функции следующим образом:

```
$ cat /proc/sys/net/ipv4/tcp_fastopen
0
```

Обычно этот параметр отключен (установлен на `0`). В табл. 9.5 перечислены некоторые варианты его значений.

Таблица 9.5. Параметры TCP Fast Open

Значение	Функция
0	Отключен
1	Включен для исходящих подключений
2	Включен для входящих подключений
3	Включен как для исходящих, так и для входящих подключений

Значение параметра вы можете изменить с помощью следующей команды:

```
echo "3" > /proc/sys/net/ipv4/tcp_fastopen
```

Помимо настройки этого параметра в операционной системе, для его использования вам следует настроить еще и серверное программное обеспечение. Например, nginx допускает наличие этого параметра на стороне веб-сервера[1], но для этого требуются параметры и конфигурации компиляции, поэтому по умолчанию параметр не включен. Windows IIS[2] поддерживает этот параметр. В документации Apache нет информации об этом параметре, поэтому, по-видимому, сервер его не поддерживает. Другие, менее распространенные серверы также могут не поддерживать его. На стороне клиента параметр можно включить в Edge[3] и Chrome на Android. Однако на момент написания этой книги в Chrome для Windows или macOS[4] он не поддерживается, а также отключен в Firefox[5].

Функция TCP Fast Open действительно очень полезна. Google заявила[6], что «на основе анализа трафика и эмуляции сети мы увидели, что TCP Fast Open снижает задержку в сети транзакций HTTP на 15 % и время загрузки всей страницы в среднем более чем на 10 %, а в некоторых случаях процент возрастает до 40 %». Однако поддержку относительно нового дополнения к TCP (RFC был опубликован в 2014 году)[7] серверы и клиенты реализовывали достаточно медленно. Учитывая все эти сложности, функция TCP Fast Open будет полезна скорее в перспективе, чем сейчас.

Использование алгоритмов управления перегрузкой, PRR и BBR

У TCP существует несколько разных алгоритмы управления перегрузкой, которые контролируют состояние TCP-соединения при потере пакетов. Большинство реализаций TCP использует алгоритм *CUBIC*[8] (по умолчанию, начиная с ядра Linux 2.6.19). Для него существует расширение, именуемое *Proportional Rate Reduction* (PRR)[9], которое при потере пакетов уменьшает размер окна управления перегрузкой вдвое (по умолчанию расширение устанавливается с версии 3.2)[10]. Подробное описание различий между всеми алгоритмами выходит за рамки этой книги. Однако стоит отметить, что правильно подобранный алгоритм может значительно повысить производительность. Для того чтобы узнать, какой алгоритм используется сейчас, воспользуйтесь следующей командой:

```
$ cat /proc/sys/net/ipv4/tcp_congestion_control
cubic
```

1 https://nginx.org/en/docs/http/ngx_http_core_module.html#listen.
2 https://blogs.technet.microsoft.com/networking/2016/07/18/announcing-new-transport-advancementsin-the-anniversary-update-for-windows-10-and-windows-server-2016/.
3 https://www.windowscentral.com/enable-tcp-fast-open-microsoft-edge-faster-page-load-times.
4 https://bugs.chromium.org/p/chromium/issues/detail?id=635080.
5 https://bugzilla.mozilla.org/show_bug.cgi?id=1398201.
6 https://ai.google/research/pubs/pub37517.
7 https://tools.ietf.org/html/rfc7413.
8 https://tools.ietf.org/html/rfc831.
9 https://tools.ietf.org/html/rfc6937.
10 https://ai.google/research/pubs/pub37486.

Список доступных алгоритмов управления перегрузкой можно получить при помщи команды:

```
$ cat /proc/sys/net/ipv4/tcp_available_congestion_control
reno cubic
```

Согласно проведенным исследованием, более новый алгоритм *Bottleneck Bandwidth and Round-trip propagation time* (*BBR*) обеспечивает еще больший прирост производительности[1], особенно для HTTP/2-соединений[2]. BBR был создан Google и доступен в ядре Linux 4.9; изменений на стороне клиента он не требует. Для его включения в Linux (версии 4.9 или новее) используйте следующие команды:

```
#Динамически загружать модуль tcp_bbr, если он еще не загружен.
sudo modprobe tcp_bbr#Add
#Добавить правило управления трафиком Fair Queue, с которым BBR лучше работает.
sudo echo "net.core.default_qdisc=fq" > /etc/sysctl.conf
#Изменить алгоритм перегрузки TCP на BBR.
sudo echo "net.ipv4.tcp_congestion_control=bbr" > /etc/sysctl.conf
#Перезагрузить настройки.
sudo sysctl -p
```

Вопреки всем положительным результатам экспериментов, некоторые исследователи[3] отмечают, что BBR плохо согласовывается с другими типами трафика и может отнимать слишком много сетевых ресурсов.

9.1.5 Будущее TCP и HTTP

Итак, я рассказал вам о некоторых сложностях TCP-протокола, который на первый взгляд кажется довольно простым, но на деле оказывается намного сложнее. Как и HTTP/1.1, TCP имеет некоторые врожденные недостатки, с которыми пользователи могут начать сталкиваться только сейчас, поскольку проблемы протоколов более высокого уровня, таких как HTTP, устранены, а требования к сетям продолжают расти.

На данный момент TCP находится в стадии развития, хотя и очень медленного. Несмотря на то что для него постоянно создаются новые параметры и алгоритмы управления перегрузкой, а браузеры регулярно обновляются, для поддержки этих функций, для того, чтобы все эти новшества вошли в сетевые стеки серверов, требуется некоторое время. Новые функции обычно связаны с фундаментальными частями операционной системы, поэтому они требуют полного ее обновления. Если в вашей операционной системе доступны новые функции, их можно включить вручную, но зачастую лучшим вариантом остается обновление. Все изменения в этой области лучше доверить разработчикам операционной системы, которые обладают необходимыми навыками и знаниями. Од-

---

1 https://cloudplatform.googleblog.com/2017/07/TCP-BBR-congestion-control-comes-to-GCP-your-Internetjust-got-faster.html.

2 https://blog.cloudflare.com/http-2-prioritization-with-nginx/.

3 https://doc.tm.uka.de/2017-kit-icnp-bbr-authors-copy.pdf.

нако, все равно стоит иметь представление обо всех упомянутых мной нововведениях, ведь в будущем они могут стать чрезвычайно полезны.

Кроме того, я коснулся далеко не всех сетевых магистралей. Даже если обе стороны поддерживают некоторые из этих функций TCP, но какие-либо промежуточные элементы инфраструктуры не поддерживают их, существует риск возникновения проблем. Подобно тому, как прокси-серверы HTTP меняют протокол соединений на HTTP/1.1, даже если оба конца поддерживают HTTP/2, инновации в области TCP могут сдерживаться так называемыми промежуточными устройствами. Поскольку TCP – это старый алгоритм, у некоторых из этих промежуточных устройств есть определенные ожидания относительно того использования, и при применении новых, неожиданных для них методов, возникают ошибки.

По этим и другим причинам некоторые люди начали задумываться, является ли TCP приемлемым для HTTP, или же стоит с нуля разработать другой, независимый от операционной сети, протокол, который будет удовлетворять потребности HTTP (текущие и будущие). Одним из таких протоколов стал QUIC.

9.2 *QUIC*

QUIC (произносится как «квик») – это новый протокол на основе UDP от компании Google (снова от Google!), призванный заменить TCP и другие части традиционного стека HTTP и тем самым устранить многие недостатки, упомянутые в этой главе. QUIC выводит концепции, представленные в HTTP/2 (например, пакеты и управление потоком) на новый уровень и заменяет TCP.

Что означает название QUIC?

Первоначально название QUIC было акронимом Quick UDP Internet Connections. Изначально под этим именем его можно было встретить в большей части документации Google Chromium[a,b,c]. При официальном оформлении рабочая группа QUIC решила отказаться от этого варианта[d], и теперь в спецификации протокола поясняется: «QUIC – это имя, а не акроним»[e]. Однако во многих источниках его все еще можно встретить под первоначальным названием. Один из членов рабочей группы забавно заметил: «QUIC – это не акроним. Скорее, это ваш удивленный возглас ;)»[f].

---

[a] https://www.chromium.org/quic.

[b] https://docs.google.com/document/d/1gY9-YNDNAB1eip-RTPbqphgySwSNS-DHLq9D5Bty4FSU/.

[c] https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/.

[d] https://github.com/quicwg/base-drafts/pull/1282.

[e] https://tools.ietf.org/html/draft-ietf-quic-transport#section-2.

[f] https://www.ietf.org/mail-archive/web/quic/current/msg03844.html.

По задумке разработчиков QUIC должен был выполнять следующие функции[1]:

- значительное сокращение времени установления соединения;
- улучшенный контроль перегрузки;
- преодоление HOL-блокировки;
- прямую коррекцию ошибок;
- миграцию соединения.

Первые три функции – это как раз то, чего недоставало TCP (и HTTPS). А следующие две – это интересные дополнения, созданные для дальнейшего решения всех этих проблем.

Прямая коррекция ошибок (Forward error correction, FEC) позволяет сократить потребность в повторной передаче пакетов за счет включения части пакета QUIC в соседние пакеты. Таким образом, если один пакет теряется, появляется возможность восстановить его по частям из доставленных пакетов. Данный метод называли «RAID 5 на сетевом уровне»[2]. Ранее я говорил, что пакеты могут теряться случайным образом, но не факт, что это следует считать причиной для ограничения соединения, и FEC стремится исправить проблему потери пакетов. Безусловно, вместе с QUIC возрастает избыточность оборудования и расходы. Однако, учитывая тот факт, что HTTP требует гарантированной доставки (в отличие, скажем, от протоколов видеопотока, где потеря пакетов обычно не приводит к каким-либо последствиям), польза от QUIC намного превышает затраты на него. На момент написания этой статьи FEC все еще является экспериментальной[3] функцией. Также она не будет включена в первоначальную версию QUIC, поскольку она не рассматривается в QUIC-WG[4].

Функция *миграции соединения* позволяет соединению перемещаться между сетями, благодаря чему снижаются расходы на установку соединения. В TCP с обеих сторон соединение было связано с IP-адресом и портом. Для изменения IP-адреса необходимо было установить новое TCP-соединение. Во времена TCP такая практика считалась приемлемой, поскольку считалось, что изменение IP-адреса в течение времени существования сеанса маловероятно. Теперь, когда существует несколько типов сетей (проводные, беспроводные и мобильные), такой метод устарел. QUIC делает возможным начать сеанс через Wi-Fi дома, а затем перейти в мобильную сеть, избежав при этом перезапуска сеанса. Также благодаря новому протоколу возможно использовать Wi-Fi и мобильную сеть одновременно для одного соединения. Такая возможность появилась благодаря методу многопутевых передач, увеличивающему пропускную способность. Хотя этот метод не будет представлен в первоначальной версии QUIC, функция миграции соединения останется.

---

1 https://www.chromium.org/quic.

2 https://ma.ttias.be/googles-quic-protocol-moving-web-tcp-udp/.

3 https://docs.google.com/document/d/1Hg1SaLEl6T4rEU9j-isovCo8VEjjnuCPT-cLNJewj7Nk.

4 https://datatracker.ietf.org/wg/quic/about/.

9.2.1 Преимущества QUIC в производительности

В апреле 2015 года Google опубликовала в своем блоге пост[1], посвященный преимуществам производительности QUIC. Он содержал следующую информацию:

- 75 % соединений теперь устанавливаются быстрее;
- в Google Search среднее время загрузки страницы сократилось на 3 %, а в медленных сетях на одну секунду. Данные цифры могут показаться небольшими, но помните, что мы стараемся оптимизировать Google Search для широкого круга пользователей, и любое улучшение является значимым;
- благодаря QUIC количество прерываний видеороликов из-за повторной буферизации на YouTube сократилось на 30 %.

Я полагаю, что результаты сравнивались с HTTP/2 и SPDY. В то время в Google QUIC использовали лишь 50 % трафика Chrome; с тех пор этот процент значительно возрос. До недавнего времени QUIC поддерживался только Chrome и Google (см. раздел 9.2.6), поэтому его использование ограничено. Согласно W3Tech на момент написания этой книги QUIC используют чуть больше 1 % сайтов[2], хотя по другим оценкам эта цифра составляет 7,8 % от объема трафика[3], из которых 98 % приходится на Google.

9.2.2 Место QUIC в стеке Internet

QUIC является альтернативой не только для TCP. На рис. 9.18 показано место QUIC в традиционном стеке технологий HTTP.

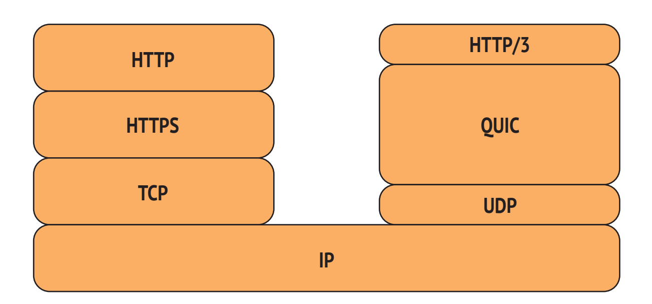


Рис. 9.18 Место QUIC в стеке технологии HTTP

Как видите, QUIC заменяет большую часть функций TCP (части настройки, надежности и управления перегрузкой), полностью заменяет HTTPS (сокращает задержки настройки) и даже часть HTTP/2 (управление потоком и части сжатия заголовка).

---

1 https://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html.
2 https://w3techs.com/technologies/details/ce-quic/all/all.
3 https://blog.apnic.net/2018/05/15/how-much-of-the-internet-is-using-quic/.

QUIC призван обеспечить возможность установки соединений с помощью одного цикла приема-передачи. Он функционирует на уровне соединения (традиционно это был TCP) и на уровне шифрования (TLS). Для этого QUIC применяет совершенно новые концепции, а также методы, которые были добавлены еще в TCP (например, Fast Open) и TLS (например, TLSv1.3).

Что касается более высокого уровня, QUIC не заменяет HTTP/2 полностью, однако он берет на себя некоторые части транспортного уровня, оставляя HTTP/2 более легкие части. Аналогично переходу от HTTP/1.1 к HTTP/2 основной синтаксис HTTP (с которым работают профессиональные разработчики) в QUIC остается неизменным; концепции, представленные в HTTP/2 (например, мультиплексирование потоков, сжатие заголовков и server push) также остались почти в неизменном виде. QUIC заботится о некоторых деталях более низкого уровня. Переход от HTTP/1.1 к HTTP/2 влечет за собой большие изменения для разработчиков, но QUIC не вносит каких-либо новых изменений, поэтому все, что вы подчерпнули в книге, не пропадет зря! Протокол по-прежнему является мультиплексированным двоичным протоколом на основе потоков, и некоторые особенности, используемые для достижения этого изменения на более низком уровне, теперь относятся к QUIC, а не к HTTP/2. Чтобы разграничить HTTP/2 и QUIC и показать преимущества нового протокола, было решено, что HTTP поверх QUIC будет называться HTTP/3 (более подробно я расскажу об этом в разделе 10.3)[1].

9.2.3 Что такое UDP и почему он является основой QUIC

QUIC основан на протоколе UDP (User Datagram Protocol), который, в сравнении с TCP, является более простым; однако он аналогичным образом построен поверх Internet Protocol (IP). TCP обеспечивает сетевому соединению надежность в IP, включая повторную передачу, перегрузку и управление потоком. Обычно эти функции полезны и необходимы, но для HTTP/2 они мало что значат. В HTTP/2 эти функции не обязательны на сетевом уровне; к тому же в TCP они создают проблему HOL-блокировки.

UDP – это базовый протокол. Он имеет концепцию портов, аналогичную концепции TCP, поэтому на одном компьютере может работать несколько служб на основе UDP. Также он имеет дополнительную функцию контрольной суммы, которая позволяет проверить целостность UDP-пакетов. За исключением этих двух функций, в протоколе нет ничего особенного. Надежности, упорядоченности и контроля перегрузки он не гарантирует, а если эти функции вам понадобятся, то они должны обеспечиваться приложением. Если UDP-пакет потерян, он не будет отправлен повторно автоматически. Если пакеты UDP приходит не по порядку, приложение более высокого уровня все равно сможет их просмотреть. Первоначально UDP использовался для приложений, которым не требовалась гарантированная доставка (например, видео, в котором неко-

---

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/2018OctDec/0065.html.

торые фреймы могли быть отброшены без особых потерь). UDP также идеально подходит для мультиплексированного протокола, такого как HTTP/2. Он обеспечит лучшее решения проблем, чем TCP.

Почему бы просто не улучшить TCP?

Самый очевидный вопрос: почему бы просто бы не улучшить TCP? TCP все еще совершенствуется, и проблемы в нем можно решить путем дальнейших улучшений. Проблема здесь заключается в скорости внедрения подобных улучшений. TCP встроен в операционную систему, и, хотя некоторые изменения могут быть внесены на стороне сервера, предпочтительнее делать это с помощью обновлений системы. Проблема не в том, что операционные системы не могут внедрять инновации, а в том, что для этого требуется очень много времени. TCP Fast Open – яркий тому пример; он предлагает значительные преимущества, но пока не получил широкой популярности у браузеров и серверов.

Затяжной процесс внедрения инноваций усугубляется инфраструктурой Internet. Ей свойственно делать определенные предположения о протоколах, таких как TCP, и, когда в реальности все происходит по-другому, возникают ошибки. Такая проблема известна как *окостенение протокола*. Отказавшись от TCP, QUIC получает больше свободы и меньше ограничений.

Почему бы не использовать SCTP?

Вместо того чтобы создавать новый транспортный протокол на базе UDP или ждать, пока инновации в TCP станут более широко распространенными, QUIC мог бы использовать *протокол передачи с управлением потоком* (Stream Control Transmission Protocol, SCTP). Данный протокол имеет много общих характеристик с QUIC, например потоковую передачу надежных сообщений. С 2007 года SCTP существует как самостоятельный стандарт.

К сожалению, статус официального стандарта не гарантирует популярности протокола. SCTP был принят не слишком тепло, поскольку существовал TCP, который прекрасно справлялся с его задачами. Следовательно, переход на SCTP может занять столько же времени, сколько и обновление TCP. И даже после такого шага совершенствование протокола, скорее всего, прекратится. QUIC стремится улучшить функцию контроля перегрузки на уровне потока и решить другие проблемы, влияющие на работу всего Internet, такие как установка HTTPS-соединений, ограниченная потеря пакетов и миграция соединения.

Почему бы не использовать IP напрямую?

Другой вариант, который могли бы использовать разработчики QUIC, – это построение его на основе IP, поскольку требования транспортного уровня обычно невелики. IP – это не что иное, как IP-адрес источника и места назначения; поверх него можно построить все, что угодно.

Однако в таких случаях возникают те же проблемы, что и при использовании SCTP. Протокол должен быть реализован на уровне операцион-

ной системы, поскольку лишь немногие приложения получают прямой доступ к IP-пакетам. Кроме того, QUIC должен быть направлен на конкретное приложение, поэтому ему нужны порты, которые есть у UDP. Многие клиенты могут открывать отдельные HTTP-соединения через QUIC, например, для одновременного запуска Chrome и Firefox, а также, возможно, еще какой-нибудь программы, использующей HTTP. Без этой функции приложение, контролирующее QUIC, должно было бы читать все пакеты QUIC и при необходимости направлять их в каждое приложение.

Преимущества UDP

UDP – это базовый протокол, который встроен в *ядро* операционной системы. Все, что построено на его основе, должно быть построено на уровне *пользовательского пространства*. Находясь за пределами ядра, можно быстро внедрять инновации за счет развертывания приложения на любой стороне. Google использует QUIC во всех своих сервисах браузера Chrome. Просмотреть текущую используемую версию QUIC (версия 43 на момент написания этой книги) вы можете, открыв инструменты разработчика и перейдя на сайт Google, как показано на рис. 9.19.

Elements Console Sources Network Performance Memory Application Security

View: Group by frame Preserve log Disable cache Offline Onli

Filter Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other

Name	Status	Protocol	Scheme	Domain
www.google.com	200	http/2+quic/43	https	www.google.com
googlelogo_color_120x44dp.png /images/branding/googlelogo/2x	200	http/2+quic/43	https	www.google.com
googlelogo_color_272x92dp.png /images/branding/googlelogo/2x	200	http/2+quic/43	https	www.google.com

Рис. 9.19. Просмотр текущей версии QUIC на www.google.com

За несколько лет существования QUIC Google создал 43 его версии[1]. Как и при развертывании SPDY, Google легко и незаметно для пользователей смог развернуть изменения в основном клиенте, используемом для просмотра веб-страниц (Chrome), и некоторых из самых популярных серверов. По состоянию на 2017 год около 7 % сервисов в Internet используют QUIC[2], хотя эту цифру, скорее всего, составляют в основном сайты Google.

Быстрая реализация QUIC была возможна только при использовании в качестве основы протокола UDP. Принудительного изменение суще-

[1] История версий подробно описана в исходном коде: https://chromium.googlesource.com/chromium/src/+/master/net/third_party/quic/core/quic_versions.h.

[2] https://ai.google/research/pubs/pub46403.

ствующих протоколов потребовало бы слишком много времени, и, вероятно, было бы заблокировано большей частью существующей инфраструктуры Internet. Использование простого протокола UDP позволило Google создавать и совершенствовать протокол по своему усмотрению, поскольку компания могла контролировать обе стороны соединения.

Конечно, у UDP тоже есть свои проблемы. Данный протокол не такой распространенный, как TCP. Например, DNS работает через UDP по причине его простоты (хотя есть шаги, позволяющие DNS работать через HTTPS, как обсуждается в главе 10). Другие приложения (например, потоковое видео в реальном времени и онлайн-видеоигры) также используют UDP, поэтому он поддерживается подавляющей частью сетевой инфраструктуры. Однако TCP гораздо более распространен, а UDP часто по умолчанию блокируется межсетевыми экранами и трафиком промежуточного программного обеспечения. В этом случае Chrome изящно возвращается к HTTP/2 через TCP. По началу это было серьезной проблемой, но эксперименты Google показали, что 93 % UDP-трафика обрабатывается нормально, и со временем этот показатель даже улучшился. Иногда инфраструктура все же блокирует трафик UDP для HTTP (где также используется порт 443), однако в большинстве случаев этого не происходит. На UDP, если он станет стандартном (как, по крайней мере, для сервисов Google), будет очень легко перейти.

Другая проблема UDP заключается в том, что пользовательское пространство бывает менее эффективным, чем оптимизированное пространство ядра. Измерения QUIC на раннем этапе показывают, что серверы используют в 3,5 раза больше ресурсов ЦП, чем эквивалентные серверы на основе TLS/TCP[1]. Несмотря на оптимизацию, исследования показывают, что UPD является более затратным протоколом и, вероятно, останется таковым до тех пор, пока он будет находиться вне ядра.

Всегда ли QUIC будет использовать UDP?

В исходном FAQ, выпущенном при запуске QUIC[a], Google заявила: «Мы надеемся, что функции QUIC будут перенесены на TCP и TLS, если они окажутся эффективными».

Так что вполне вероятно, что объектом экспериментов в ближайшее время станет UDP, а TCP будет развиваться более медленными темпами. Вернется ли QUIC к TCP? На этот вопрос сложно ответить, но я считаю, что отказаться от свободы развития Google будет сложно. Internet, похоже, находится в фазе инноваций транспортного уровня; сложно представить, что разработчики протоколов захотят прекратить вводить новшества и перейти к фиксированному протоколу, который будет сложно обновить.

Кроме того, по сравнению с TCP, QUIC вносит фундаментальные изменения, которые будет нелегко внедрить в TCP, даже если бы для этого существовал стимул.

1 https://dl.acm.org/citation.cfm?id=3098842.

Более вероятно, что HTTP по-прежнему будет доступен как для TCP (HTTP/2), так и для UDP (QUIC и HTTP/3), но реализация TCP будет отставать от UDP с точки зрения функций и производительности.

[a] https://docs.google.com/document/d/1lmL9EF6qKrk7gbazY8bIdvq3Pno2Xj_l_YShP40GLQE/.

9.2.4 Стандартизация QUIC

QUIC был представлен компанией Google в июне 2013 года[1]. На протяжении следующих двух лет Google занималась доработкой протокола и в июне 2015-го предложила его Инженерной группе Internet (IETF) в качестве стандарта[2]. Google подала протокол на рассмотрения в крайне удачный момент, ведь предыдущий стандарт компании (SPDY) был официально принят как HTTP/2; люди, связанные с разработкой HTTP/2 могли работать и над QUIC. Несколькими месяцами позже для разработки и оформления стандарта нового протокола была создана рабочая группа IETF QUIC[3].

Два варианта QUIC: gQUIC и iQUIC

Как и SPDY, QUIC развивался под руководством Google параллельно с разработкой стандарта. Ввиду этого на момент написания этой книги существует две реализации протокола gQUIC (Google QUIC) и iQUIC (IETF QUIC). Рабочая среда Google использует gQUIC; компания продолжает развивать и улучшать ее на свое усмотрение, без необходимости официального подтверждения каждого изменения.

Ожидается, что gQIUC, подобно SPDY, выйдет из использования после стандартизации iQUIC (скорее всего, это произойдет в начале 2019 года). Однако на данный момент gQUIC является единственной версией протокола, которую можно использовать в рабочих средах.

QUIC (а именно gQUIC) используется только Chrome и браузерами на базе Chromium, например Opera. В свою очередь gQUIC претерпевает частые изменения по мере внедрения в него нововведений командой Google[4]. На серверной стороне его поддерживают все сервисы Google. Другие реализации веб-серверов на момент написания этой книги используют Caddy[5] и LiteSpeed[6], но, поскольку они основаны на развивающемся нестандартизированном gQUIC, эти реализации меняются вместе

1 https://blog.chromium.org/2013/06/experimenting-with-quic.html.

2 https://datatracker.ietf.org/doc/draft-tsvwg-quic-protocol/00/.

3 https://datatracker.ietf.org/wg/quic/about/.

4 См. раздел «Последние изменения версии» на странице https://docs.google.com/document/d/1WJvyZflAO2pq77yOLbp9NsGjC1CHetAXV8I0fQe-B_U/.

5 https://github.com/mholt/caddy/wiki/QUIC.

6 https://blog.litespeedtech.com/2017/07/11/litespeed-announces-quic-support/.

с ним, и в случае, если обновления задерживаются, реализации могут перестать работать в Chrome[1].

Различия между gQUIC и iQUIC

По мере развития каждого из протоколов различия становятся все ощутимее, однако на момент написания этой книги одно из основных различий между gQUIC и iQUIC заключается в уровне шифрования. Google применяет специализированный способ криптографии, а iQUIC работает с TLSv1.3[2]. Выбор Google обусловлен тем, что на момент создания QUIC TLSv1.3 попросту не существовало. Google заявила, что заменит свой способ криптографии на TLSv1.3, когда тот будет признан официально[3], что на данный момент уже произошло, поэтому gQUIC и iQUIC постепенно сходятся к одной точке. Между этими двумя протоколами существует несколько других различий, и в этих точках они несовместимы, но на концептуальном уровне, за исключением использования TLSv1.3, они похожи.

Стандарты QUIC

На данный момент у QUIC нет одного общепринятого стандарта, зато есть целых шесть отдельных. Как и HTTP/2, который состоит из двух стандартов (HTTP/2 и HPACK), QUIC имеет отдельные стандарты для своих основных частей:

- *QUIC Invariants*[4] – стандарт для неизменных частей протокола;
- *QUIC Transport*[5] – основной транспортный протокол;
- *QUIC Recovery*[6] – контроль потери пакетов и перегрузки;
- *QUIC TLS*[7] – применение шифрования TLS в QUIC;
- *HTTP/3*[8] – протокол, в значительной степени основан на HTTP/2 с некоторыми изменениями;
- *QUIC QPACK*[9] – сжатие HTTP-заголовков в QUIC.

Также был предложен еще один экспериментальный документ под названием *QUIC Spinbit*[10], который добавлял бы отдельный бит, используемый при мониторинге зашифрованных соединений QUIC. Кроме того, доступны два дополнительных информационных документа по исполь-

---

1 https://github.com/mholt/caddy/issues/2194.
2 https://tools.ietf.org/html/rfc8446.
3 https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5L-TaDUDwvZ5L6g.
4 https://tools.ietf.org/html/draft-ietf-quic-invariants.
5 https://tools.ietf.org/html/draft-ietf-quic-transport.
6 https://tools.ietf.org/html/draft-ietf-quic-recovery.
7 https://tools.ietf.org/html/draft-ietf-quic-tls.
8 https://tools.ietf.org/html/draft-ietf-quic-http.
9 https://tools.ietf.org/html/draft-ietf-quic-qpack.
10 https://tools.ietf.org/html/draft-ietf-quic-spin-exp.

зованию QUIC: для разработчиков приложений[1] и для распространения QUIC в сети[2].

На данный момент IEFT еще работает над этими документами. Поскольку стандарт все еще находится в стадии разработки, эти спецификации могут претерпеть значительные изменения, а также может измениться их количество.

Следует отметить один важный момент: QUIC задуман как протокол общего назначения; HTTP – это только одна из областей, где его можно применить. На сегодняшний день HTTP является основным вариантом использования QUIC (чем преимущественно и занимается рабочая группа), протокол все же разрабатывается с учетом других возможных вариантов использования.

9.2.5 Различия между HTTP/2 и QUIC

QUIC основан на HTTP/2, поэтому многие из основных концепций, представленных в этой книге, пригодятся, когда QUIC станет официальным стандартом и выйдет за пределы серверов и браузеров Google. Однако существуют и различия, например HTTP/2 и QUIC созданы на базе разных протоколов. В разделах ниже я расскажу и о других различиях.

QUIC и HTTPS

HTTPS встроен в QUIC и, в отличие от HTTP/2, это закрывает незашифрованным HTTP-соединениям доступ к QUIC. Такой вариант был выбран по тем же идеологическим и практическим причинам, что и в HTTP/2, просмотр страниц в котором доступен только через HTTPS (см. главу 3).

С практической точки зрения шифрование данных гарантирует, что сторонние элементы инфраструктуры не будут невольно вмешиваться в его работу. Сейчас эта ситуация может показаться не такой уж серьезной (ведь ни один элемент инфраструктуры не будет работать с UPD через HTTP-трафик), однако она уже вызвала проблемы для QUIC, поскольку промежуточные устройства, в сравнении с QUIC, несколько устарели[3]. По мере развития протокола становится еще более важным аспект предотвращения окостенения, что мы можем наблюдать в случае с TCP по причине использования промежуточных устройств, управляющих трафиком TCP. QUIC стремится к шифровке наибольшего количества данных. Предложение разрешить использование одного незашифрованного бита, позволяющего промежуточным устройствам отслеживать трафик[4], было принято негативно[5], и на момент написания этой книги разработ-

---

1 https://tools.ietf.org/html/draft-ietf-quic-applicability.

2 https://tools.ietf.org/html/draft-ietf-quic-manageability.

3 https://datatracker.ietf.org/doc/draft-ietf-quic-spin-exp/.

4 https://news.ycombinator.com/item?id=16695816.

5 https://news.ycombinator.com/item?id=16695816.

чики не пришли к однозначному решению (хотя предложение находится в стадии рабочего проекта, как упоминалось ранее в этой главе).

Установка QUIC-соединения

В HTTP/2 существовало несколько методов согласования протокола: ALPN, заголовок `Upgrade`, использование заранее известного протокола и HTTP-заголовок или HTTP/2-фрейм Alt-Svc. Все они предполагают использование TCP. Поскольку QUIC основан на UDP, веб-браузер, подключающийся к веб-серверам, должен запускать соединение по TCP и затем переходить до QUIC[1]. Такой процесс вводит QUIC в зависимость от HTTP и, следовательно, сводит на нет одно из его ключевых преимуществ (значительное сокращение времени установки соединения). В качестве альтернативы можно использовать TCP и UDP одновременно или же смириться со скачками производительности при установке соединения, возможно, вспомнив в следующий раз, что сервер использует QUIC. В любом случае для HTTP/3 будет установлен идентификатор ALPN и Alt-Svc `h3` (примечание: до того, как было принято имя HTTP/3, идентификатором для HTTP через QUIC был `hq`). Данный идентификатор следует использовать только для стандартизированного iQUIC; существующие реализации gQUIC не должны использовать это значение[2].

QPACK

Алгоритм сжатия заголовков HPACK в силу характера TCP гарантирует упорядоченное получение фреймов HTTP-заголовков, чтобы динамическая таблица поддерживалась обеими сторонами, как показано на рис. 9.20.

Запрос 2 использует индексы заголовка, определенные в запросе 1 (62 и 63). При потере части запроса 1 заголовок не может быть прочитан полностью, поскольку состояние динамической таблицы не известно. Таким образом, запрос 2 не может быть обработан до тех пор, пока не будут получены отсутствующие пакеты, иначе это может привести к использованию неверных ссылок. QUIC стремится устранить необходимость в упорядоченной доставке пакетов на уровне соединения, чтобы потоки могли обрабатываться независимо. HPACK требует именно этого (по крайней мере, для кадров заголовков), ввиду чего возникает проблема HOL-блокировки, с которой призван бороться QUIC.

Именно поэтому HTTP/3 нуждался в обновленном варианте HPACK; его создали и по понятным причинам назвали QPACK. Данный алгоритм довольно сложен, и на момент написания этой книги его все еще дорабатывают. Однако именно он вводит концепцию подтвержденных заголовков. Если отправителю необходимо использовать неподтвержденный заголовок, он может использовать его (и тогда его, возможно, заблокируют в этом потоке) или может отправить заголовок с литеральными константами (что предотвратит блокировку за счет менее эффективного сжатия для этого значения заголовка).

---

1 https://tools.ietf.org/html/draft-ietf-quic-http-12#section-2.1.

2 https://github.com/w3c/navigation-timing/issues/71.

QPACK отличается от HPACK еще по нескольким пунктам. В нем бит определяет, используется ли статическая или динамическая таблица (в то время как в HPACK отсчет начинается с 61). Кроме того, в QUIC заголовки можно легко и эффективно дублировать; это делается для того, чтобы заголовки ключей (такие как `Authority` и `user-agent`) оставались в верхней части динамической таблицы и передавались посредством меньшего количества битов.

Статическая таблица HPACK

Значение индекса	Имя заголовка	Значение заголовка
1	:authority	
2	:method	GET
3	:method	POST
4	:path	/
...	...	...
58	user-agent	
...	...	...
61	www-authenticate	

Запрос 1

Заголовок	Значение заголовка
:method	GET
:authority	www.example.com
:path	/
user-agent	Chrome-69

Сжатый запрос 1

Заголовок	Значение заголовка
Индексировано 2	
Литеральный индекс 24 с индексацией	www.example.com
Индексировано 4	
Литеральный индекс 56 с индексацией	Chrome-69

Динамическая таблица после запроса 1

Значение индекса	Имя заголовка	Значение заголовка
62	user-agent	Chrome-62
63	:authority	www.example.com

Запрос 2

Заголовок	Значение заголовка
:method	GET
:authority	www.example.com
:path	/styles.css
user-agent	Chrome-69

Сжатый запрос 2

Заголовок	Значение заголовка
Индексировано 2	
Индексировано 63	www.example.com
Литеральный индекс 4 без индексации	
Индексировано 62	Chrome-69

Динамическая таблица после запроса 2

Значение индекса	Имя заголовка	Значение заголовка
62	user-agent	Chrome-62
63	:authority	www.example.com

Рис. 9.20 Пример сжатия HPACK

Другие различия

В QUICK были внесены еще несколько изменений касаемо функционирования фреймов и потоков[1]. От уровня HTTP/3 отделяют некоторые фреймы протоколов транспортного уровня (например, фреймы `PING`

[1] https://github.com/quicwg/wg-materials/blob/master/interim-18-06/HTTP.pdf.

и WINDOW_UPDATE). Они перемещаются на базовый уровень QUIC-Transport, не связанный с HTTP (что имеет смысл, поскольку эти фреймы, вероятно, будут использоваться для протоколов, отличных от HTTP). Также разработчики отказались от фрейма CONTINUATION, который в HTTP/2 практически не использовался. Еще произошли некоторые изменения в плане форматирования фреймов, однако, поскольку протокол еще находится в стадии развития, я считаю, что не стоит обсуждать их сейчас. По сути, QUIC во многих аспектах похож на HTTP/2, поэтому, усвоив информацию о предыдущей версии, читатели будут иметь более или менее четкое представление о QUIC и HTTP/3, когда они будут официально стандартизированы и станут доступными для клиентских и серверных реализаций.

9.2.6 Инструменты QUIC

Поскольку QUIC еще не стандартизирован, для пользователей доступен только gQUIC, хотя многие разработчики все же работают над реализациями iQUIC[1]. Наилучшим инструментом для работы с QUIC является Chrome, когда он соединен с сервером Google. В QUICK, также как в HTTP/2, существует страница net-export (см. раздел 4.3.1). Кликнув на строку сеанса QUICK, вы увидите окно наподобие показанного на рис. 9.21.

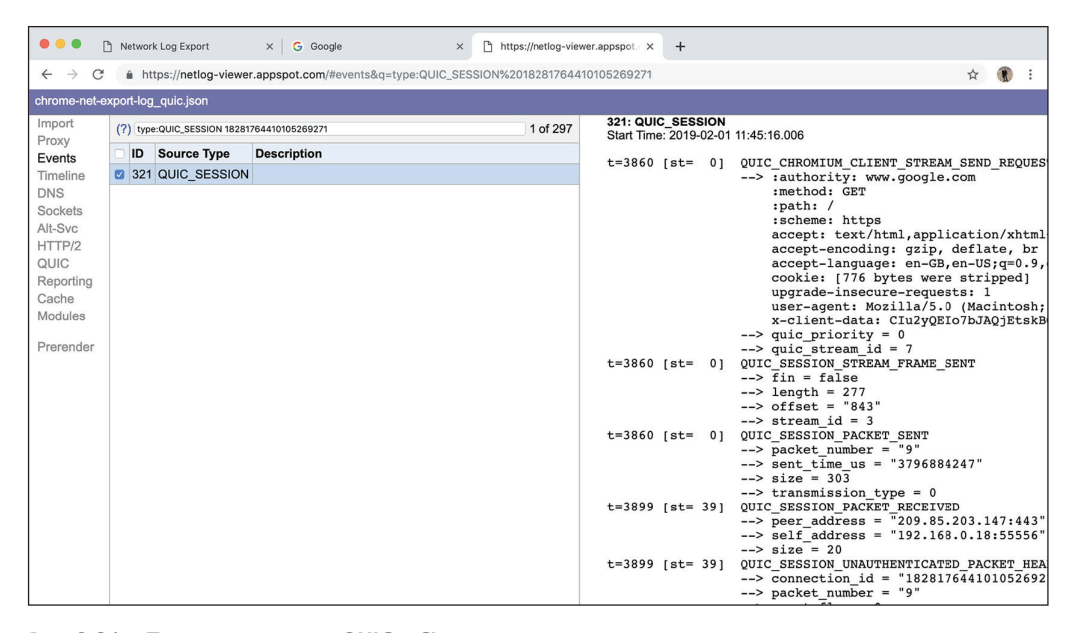


Рис. 9.21 Просмотр данных QUIC в Chrome

Другие инструменты, такие как Wireshark, поддерживаются gQUIC, как видно на рис. 9.22.

[1] https://github.com/quicwg/base-drafts/wiki/Implementations.

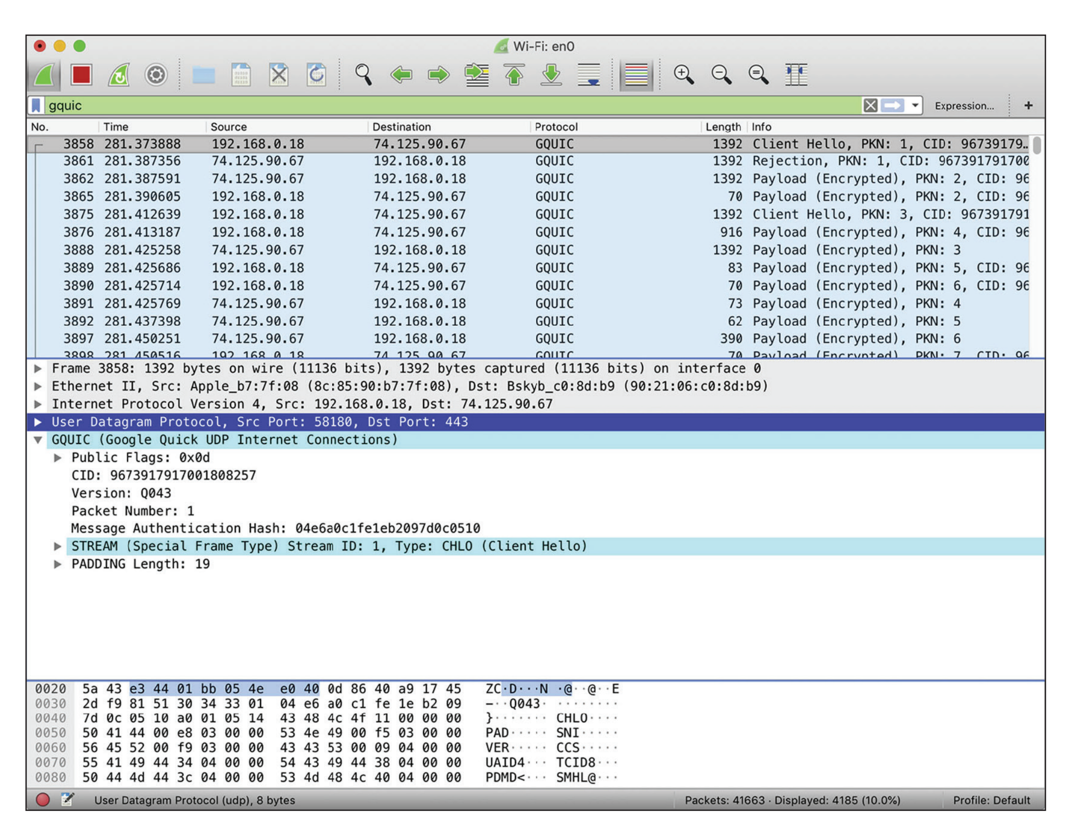


Рис. 9.22 **gQUIC в Wireshark**

Поскольку gQUIC не стандартизирован и все еще находится в стадии доработки, он должен идти в ногу с любыми изменениями, которые вносит Google. По моему опыту, при работе с ним вы можете обнаружить искаженные пакеты или зашифрованные полезные данные, которые по этой причине не могут быть прочитаны сторонними инструментами.

9.2.7 Реализации QUIC

При создании реализаций QUIC история повторяется. У Caddy была реализация gQUIC, основанная на реализации QUIC на языке программирования Go, однако на данный момент от нее отказались[1]. Она будет доступна благодаря компиляции Caddy из исходного кода и должна появиться в следующей версии. Версия Go[2] часто обновляется, поэтому, если вы загрузите последнюю версию, Chrome должен иметь возможность взаимодействовать с ней по gQUIC. LiteSpeed также представил реализацию QUIC[3]. Она была представлена в июне 2017 года и сейчас

---

1 https://github.com/mholt/caddy/issues/2190.

2 https://github.com/lucas-clemente/quic-go.

3 https://blog.litespeedtech.com/2017/06/26/litespeed-is-powered-by-quic/.

еще дорабатывается, однако версия с открытым исходным кодом еще не поддерживает ее, поэтому это не лучший инструмент для экспериментов с QUIC, если вы не пользуетесь LiteSpeed. LiteSpeed также открыл исходный код клиента QUIC[1], который иногда может быть крайне полезен. Совсем недавно, в мае 2018 года, о поддержке gQUIC на своей сетевой платформе доставки контента объявила компания Akamai[2], а в июне 2018 года Google объявила о поддержке gQUIC для своего балансировщика нагрузки Google Cloud Platform[3], так что те, кто использует эту платформу, получили gQUIC буквально из первых рук.

9.2.8 Стоит ли переходить на QUIC?

В отличие от SPDY gQUIC не был тепло принят большей частью широкой публики. Вероятно, с iQUIC такого не произойдет. На данный момент я не могу однозначно рекомендовать вам перейти на QUIC, за исключением случаев использования облачной платформы Google. Для всех, кто хочет поэкспериментировать с QUIC, наилучшим вариантом будет делать это с помощью Go, однако его не следует использовать в рабочей среде для браузеров. Реализация браузера Chrome может немного измениться, ведь Chrome отключает старые версии gQUIC в браузерах сразу после развертывания новых версий.

Я считаю, что после стандартизации iQUIC появятся новые реализации. На данный момент их намного меньше в сравнении с SPDY. Скорее всего, развертывание QUIC и HTTP/3 займет больше времени, чем развертывание HTTP/2, поскольку в нем гораздо больше изменений, а также он основан на UDP, а не TCP. QUICK имеет серьезные перспективы. Вероятно, что через несколько лет он будет использоваться так же широко, как HTTP/2. Благодаря Google, а также CND, обслуживающих большую часть трафика, повсеместный переход на QUIC может значительно ускориться, однако какой-то процент небольших компаний и серверов останется на TCP и HTTP/2 (или даже HTTP/1.1).

Резюме

- Сетевой стек HTTP имеет несколько недостатков на уровнях TCP и HTTPS.
- Ввиду весьма осторожного алгоритма контроля перегрузки при установке TCP-соединения требуется много времени для достижения максимальной пропускной способности, к тому же процесс замедляет квитирование HTTPS.

---

[1] https://github.com/litespeedtech/lsquic-client.

[2] https://community.akamai.com/customers/s/article/FAQ-QUIC-Native-Platform-Support-for-Media-DeliveryProducts?language=en_US.

[3] https://cloudplatform.googleblog.com/2018/06/Introducing-QUIC-support-for-HTTPS-load-balancing.html.

- Существуют новые технологии, устраняющие эту проблему, однако их внедрение на стороне TCP происходит слишком медленно.
- QUIC – это новый протокол, основанный на UDP.
- Благодаря UDP, в QUIC инновации внедряются намного быстрее, чем TCP.
- QUIC построен на базе HTTP/2, он включает в себя как уже существующие концепции, так и совершенно новые.
- QUIC предназначен не только для HTTP; в будущем он может использоваться и для других протоколов.
- HTTP, реализованный через QUIC, будет называться HTTP/3.
- У QUIC есть две версии: Google Quic (gQUIC) с ограниченным доступом и ETF QUIC (iQUIC), который в настоящее время находится в процессе стандартизации.
- Ожидается, что gQUIC будет заменен iQUIC (по завершении разработки и оформления его стандарта), так же как HTTP/2 в свое время заменил SPDY.

10 Дальнейшее развитие HTTP

В этой главе мы рассмотрим:

- споры, связанные с HTTP/2;
- работу с HTTP/2-сервером после его запуска;
- HTTP вне HTTP/2;
- HTTP как базовый транспортный уровень.

Спецификация HTTP/2 была опубликована в мае 2015 года, почти через 20 лет после выхода HTTP/1.0, который вскоре был заменен HTTP/1.1. За это время Internet стал неотъемлемой частью жизни каждого человека, и успех HTTP/1.1 многое говорит нам о качестве протокола. Однако постепенно этот протокол начал отставать от ритма современной жизни, а все попытки его инноваций[1] ограничивались лишь уточнением документации или введением новых функцией ограниченного характера, реализуемых посредством HTTP-заголовков.

На смену HTTP/1 пришел HTTP/2, которым сегодня активно пользуется большая часть пользователей[2]. Однако что ждет HTTP в будущем? Что на самом деле представляет из себя работа с HTTP/2 на практике? До конца ли решены проблемы HTTP? Сколько нам придется ждать появления новой версии: 20 лет или намного меньше? В этой главе я попытаюсь ответить на все эти вопросы, а также поделюсь с читателями некоторыми обоснованными предположениями о развитии HTTP.

---

1 https://www.w3.org/Protocols/HTTP-NG/Activity.html.

2 https://w3techs.com/technologies/details/ce-http2/all/all.

10.1 Споры о HTTP/2 и его недостатках

На протяжении всего процесса стандартизации HTTP/2 возникало большое количество разногласий, особенно сильно ситуация обострилась, когда процесс приближался к завершению и ратификации. Одни аргументы были убедительными, другие – не очень[1]. Одни считали, что SPDY не следует использовать как основу для HTTP/2, а другие говорили о нерешенных проблемах конфиденциальности. Кроме того, были высказаны аргументы за и против применения в протоколе техники принудительного шифрования. В последующих разделах я расскажу об этих и многих других спорных моментах, касающихся HTTP/2.

Множество споров возникало в списке рассылки рабочей группы HTTP (HTTP-WG) Инженерной группы Internet[2], а также в более широком Internet-сообществе на таких сайтах, как HackerNews[3,4,5], SlashDot[6] и The Register[7]. В свою очередь разработчиками протокола было выдвинуто множество контраргументов.

Сейчас, когда протокол приобрел устойчивое положение и обсуждается уже его будущее, стоит оглянуться назад и еще раз рассмотреть каждый из возникших в то время споров. Таким образом, мы сможем понять, какие проблемы на сегодняшний день остаются актуальными и что полезного из этого могут извлечь разработчики.

10.1.1 Споры о SPDY

HTTP/2 основывается на протоколе SPDY, разработанном компанией Google. SPDY – протокол прикладного уровня, который можно было внедрить и развернуть в Internet. Успех SPDY побудил IETF рассмотреть вопрос об обновлении HTTP[8]. Разумеется, IETF не ограничивался только лишь SPDY, однако именно этот протокол послужил основой для HTTP/2. Многих людей не устроило, что самому HTTP/2 уделялось несколько меньше внимания, чем SPDY.

Был ли SPDY единственным вариантом для HTTP/2?

В отношении HTTP/2 в уставе рабочей группы HTTP[9] сказано следующее:

> *Мы стремимся создать протокол, который сохранит семантику HTTP, но не унаследует структуру и синтаксис сообщений*

---

1 https://lists.w3.org/Archives/Public/ietf-http-wg/2015JanMar/0043.html.
2 https://lists.w3.org/Archives/Public/ietf-http-wg/.
3 https://news.ycombinator.com/item?id=8850059.
4 https://news.ycombinator.com/item?id=9022470.
5 https://news.ycombinator.com/item?id=9066379.
6 https://tech.slashdot.org/story/15/01/09/0118226/http2---the-ietf-is-phoning-it-in.
7 https://www.theregister.co.uk/2015/02/18/http2_specification_approved/.
8 https://lists.w3.org/Archives/Public/ietf-http-wg/2012JanMar/0098.html.
9 https://datatracker.ietf.org/wg/httpbis/charter/.

> *предыдущих версий. Опыт реализации прошлых версий показал, что методы, используемые в них, снижают производительность и приводят к неправильному функционированию лежащего в основе транспортного протокола.*
>
> *Путем внедрения концепции упорядоченных двунаправленных потоков рабочая группа создаст абсолютно новую форму уже существующей семантики HTTP. Как и для предыдущих версий, приоритетной основой станет TCP, однако возможно использование и других транспортных протоколов.*
>
> *Отправной точкой работ стал draft-mbelshe-httpbis-spdy-00.*

Данное заявление не оставляло сомнений в том, что в основу HTTP/2 должен был лечь SPDY. Несмотря на его успех, многие люди сомневались, что он пригоден для того, чтобы стать основой новой версии. Некоторые считали, что IETF следует расширить свои взгляды на совершенствование HTTP и отойти от старых структур. Также был предложен независимый план-график сроком на два года[1], и многие верили, что только его соблюдение сделает SPDY приемлемым вариантом.

Рабочей группой также были рассмотрены два других предложения: протокол HTTP Speed and Mobility от Microsoft[2] (основанный на SPDY и WebSockets[3]) и Network Friendly HTTP Upgrade[4]. Во многом оба предложения были схожи с SPDY (что не удивительно, учитывая параметры, в которых был определен HTTP/2). Также они оба были направлены на введение уровня двоичного фрейминга и усовершенствование HTTP-заголовков.

Однако в конечном итоге SPDY все же превзошел всех потенциальных конкурентов. Сегодня его поддерживает не только Google, но и другие популярные веб-серверы и веб-браузеры; многие сайты уже перешли на SPDY или находятся в процессе перехода на него.

Facebook представил предварительный анализ развития для каждого предложения[5] и выяснил, что SPDY все же является предпочтительным вариантом. Таким образом, в процессе стандартизации взятый за основу SPDY был изменен и улучшен. HTTP/2 – это не SPDY, однако они имеют немало общего.

Основная проблема заключалась в том, что возможность выйти за рамки SPDY была упущена. SPDY был разработан для решения одной серьезной проблемы с производительностью HTTP/1 – и, хотя он справлялся с этой задачей хорошо, он не решал другие проблемы, связанные с HTTP, например касающиеся файлов куки. Учитывая, что предыдущие обновления, такие как HTTP-NG, потерпели неудачу в значительной степени

---

1 https://lists.w3.org/Archives/Public/ietf-http-wg/2012OctDec/0003.html.

2 https://tools.ietf.org/html/draft-montenegro-httpbis-speed-mobility.

3 https://blogs.msdn.microsoft.com/interoperability/2012/03/25/speed-and-mobility-an-approach-for-http2-0-to-make-mobile-apps-and-the-web-faster/.

4 https://tools.ietf.org/html/draft-tarreau-httpbis-network-friendly.

5 https://lists.w3.org/Archives/Public/ietf-http-wg/2012JulSep/0251.html.

из-за того, что их область применения стала слишком обширной и внедрить их на прикладном уровне было довольно сложно, разработчикам следовало выбрать одну практическую реализацию. SPDY стал стимулом для обновления HTTP, и, если бы не он, мы, вероятно, использовали бы HTTP/1.1 и по сей день.

SPDY и Google

Помимо прочего, многих беспокоило что SPDY является результатом деятельности одной компании: Google. Несмотря на то что им пользовались различные веб-сайты разной величины (включая Yahoo!, Twitter и Facebook), он разрабатывался Google и принадлежал только этой компании. Некоторые люди негодовали, что Google, уже имеющая значительное влияние в сети, продвигает протокол собственного производства и, следовательно, хочет выйти на уровень более широкого веб-сообщества. Некоторые основания для этого недоверия имелись, поскольку компания действительно является одной из главных в веб-индустрии. Кроме того, большую часть дохода Google составляет веб-реклама, которая неблагоприятным образом может сказываться на конфиденциальности данных и отслеживании пользователей. HTTP/2 отличился тем, что он не был нацелен на решение проблем конфиденциальности, которые многие люди считали приоритетной. Однако я не считаю, что под отстранением от этих проблем компания скрывала злонамеренные цели. Я думаю, что в Google всего-навсего выбрали производительность приоритетной задачей.

Данный аргумент, однако, упускает из виду ключевое значение стандартизации SPDY для HTTP/2: отказ от зависимости от одной компании, ввиду чего веб-сообщество и сообщество интернет-стандартов (IEFT) смогут пересматривать и улучшать протокол. Google продолжает оставаться одним из ведущих новаторов в Internet. Компании принадлежит множество других достижений в области веб-стандартов, о некоторых из которых (например, QUIC; см. главу 9) я рассказываю в этой книге. Я считаю нецелесообразным игнорировать нововведения Google, поскольку большинство из них крайне полезно.

10.1.2 Проблемы конфиденциальности и состояния в HTTP

Одним из объектов споров в отношении HTTP/2 была конфиденциальность, особенно это касалось файлов куки HTTP. Ввиду негативного влияния на безопасность и конфиденциальности их зачастую называют одной из самых больших проблем HTTP. По задумке разработчиков, HTTP – это протокол, не предусматривающий сохранение состояний. В HTTP/2 этот аспект не изменился. Получается, что любой запрос к серверу не должен быть связан с предыдущими или будущими запросами.

Однако на практике выясняется, что многие современные приложения и веб-сайты требуют наличия в протоколе концепции состояния. Например, когда вы добавляете что-то в корзину на Amazon, покупка оста-

ется там, несмотря на ваши последующие действия на странице. Также, когда вы входите в систему интернет-банкинга, вам не нужно входить в систему при каждом последующем действии (по крайней мере, во время этого сеанса). Следовательно, ввиду отсутствия в HTTP связей между запросами возникла необходимость добавить этот аспект в протокол, а поскольку соединения не всегда сохранялись или использовались повторно, данная функция не могла быть реализована на уровне соединения.

У этой проблемы есть несколько решений. Например, вы можете добавлять в URL-адреса параметры идентификатора сеанса (например, http://www.example.com?SESSIONID=12345), однако они выглядят громоздко, могут запутать пользователя, а также несут в себе угрозу безопасности, поскольку такие URL могут быть использованы или добавлены в закладки другими пользователями. Решить эту проблему были призваны файлы куки[1]. Они представляют собой небольшие фрагменты информации, хранящиеся в браузере и отправляемые им при каждом запросе. Файлы куки позволяют использовать в HTTP идентификаторы сеансов или другие настройки. В последнее время репутация этих файлов в глазах пользователей сильно упала, однако на самом деле они носят нейтральный характер, а использование их альтернатив (например, параметры URL-адреса) зачастую влечет за собой более серьезные проблемы.

Файлы куки заслужили дурную репутацию по нескольким причинам, включая следующие:

- они позволяют отслеживать пользователя в целях внедрения рекламы (а иногда и более страшных вещей);
- по умолчанию они небезопасны;
- они отправляются с каждым запросом.

Куки и стороннее отслеживание

Файлы куки могут использоваться не только сайтом, но и любым ресурсом, который веб-браузер загружает для этого сайта. Использование так называемых *сторонних файлов куки* возможно, когда веб-сайт (например, www.example.com) загружает контент с рекламного сайта (например, adwords.google.com) и устанавливает файл куки, который можно использовать на других веб-сайтах (например, www.example2.com). Такой куки также ссылается на сторонний рекламный веб-сайт (adwords.google.com), как показано на рис. 10.1.

На основе истории просмотров при посещении других веб-сайтов на них может отображаться соответствующая реклама, и часто это происходит без ведома пользователя. В результате Европейский союз (ЕС) ввел в действие так называемый закон о файлах куки, согласно которому веб-сайты должны информировать пользователей о том, что они используют файлы куки, как показано на рис. 10.2.

---

[1] https://tools.ietf.org/html/rfc6265.

www.example.com
1. Запрос страницы
2. Страница с контентом adwords.google.com
3. Запрос содержания объявления
4. Объявление плюс куки
adwords.google.com
www.example2.com
5. Запрос страницы
6. Страница с контентом adwords.google.com
7. Запрос содержания объявления (на этот раз с куки)
8. Индивидуальная реклама на основе других посещенных сайтов (например, www.example.com)
adwords.google.com

Рис. 10.1 Отслеживание сторонними файлами куки

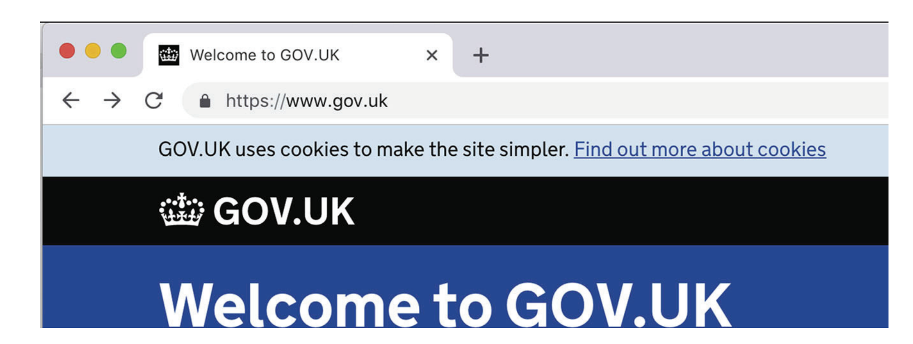


Рис. 10.2 Предупреждение об использовании куки на веб-сайте правительства Великобритании

Веб-пользователям (особенно в ЕС) приходилось скрывать сообщение «Этот веб-сайт использует файлы куки» на каждом сайте, поскольку большинство из них использовало файлы куки. В 2018 году, благодаря *Общему регламенту по защите* данных (General Data Protection Regulation, GDPR), в силу вступило еще более жесткое требование, согласно которому эти предупреждения должны были быть намного больше. Однако для большинства пользователей предупреждения – это всего лишь всплывающие окна, которые необходимо срочно закрыть, чтобы они не мешали, и вряд ли они задумываются о том, что GDPR, который является законом о защите прав потребителей и направлен на сохранение данных пользователя, вообще существует.

HTTP-куки и безопасность

Другая проблема заключается в том, что файлы куки по умолчанию небезопасны. Атрибуты, призванные решить эту проблему (например, флажки `Secure1` и `HttpOnly2`), крайне неудобны в использовании, поскольку требуют, чтобы программисты устанавливали их при создании файлов куки. И даже в случае, если они это сделают, данный способ не гарантирует предотвращения, например, перезаписи безопасных файлов куки небезопасными. На момент написания этой книги исследования показывают, что эти параметры встроены примерно в 8 % файлов куки[3], что свидетельствует о том, что подавляющее большинство файлов куки по умолчанию небезопасно.

Это плохо, потому что файлы куки, в которых хранятся идентификаторы сеанса, предлагают полный доступ к учетной записи, поэтому перехват файла куки практически аналогичен перехвату имени пользователя и пароля. По этой причине вы могли подумать, что файлы куки будут отключены (и их можно отключить), но по умолчанию это не так.

Файлы куки отправляются как с HTTP, так и с HTTPS-запросами (если при создании куки не используется атрибут `Secure`). Ввиду затрат на установку HTTPS-соединений (что по большей части больше не проблема) их используют лишь на страницах входа на сайт или на странице оформления заказа, а остальная часть сайта обслуживается HTTP. Таким образом, файлы куки без атрибута `Secure` для отправки (и, возможно, чтения) передаются по незащищенному HTTP-трафику. Точно так же иметь доступ к файлам куки (если не установлен флажок `HttpOnly`) может любой фрагмент JavaScript, загруженный на страницу (например, удобный скрипт, который делает страницу более симпатичной, добавляющий систему комментирования или классный виджет).

Даже если вы пользуетесь флажками, злоумышленники могут подделать HTTP-запрос и перезаписать безопасные файлы куки `Secure` и `HttpOnly`. Для решения этой проблемы были добавлены префиксы фай-

[1] https://tools.ietf.org/html/rfc6265#section-5.2.5.

[2] https://tools.ietf.org/html/rfc6265#section-5.2.6.

[3] https://github.com/mikewest/http-state-tokens#a-problem.

лов куки[1]. Согласно этой концепции файл куки с именем, начинающимся с `__Secure`, должен иметь атрибут `Secure`. Опять же, это решение требует участия владельцев веб-сайтов (для использования) и браузеров (для побуждения к использованию).

Файлы куки отправляются с каждым запросом

Файлы куки отправляются с каждым запросом. Безусловно, это очень просто, однако в других аспектах это порождает множество проблем. Конечно, запросы к вашей службе интернет-банкинга для проверки вашего баланса или перевода средств должны требовать отправки информации о сеансе, однако запросы логотипов или других статических ресурсов не должны нуждаться в этой информации, но браузер все равно отправляет ее. Все это приводит к возникновению проблем безопасности, а также делает веб-сайт уязвимым для атак и подделки межсайтовых запросов[2], что может привести к утечке информации. Для того чтобы отказаться от передачи куки при навигации на сайте, был предложен атрибут `SameSite3`. Однако он не является значением по умолчанию, и для его реализации необходимо обладать расширенными знаниями и навыками.

Другие типы методов отслеживания конфиденциальной информации часто называются куки, даже если они не являются HTTP-куки. *Flash cookie*, которые реализованы во Flash, или *супер куки*, используют для отслеживания пользователей методы снятия цифрового отпечатка[4]. Также существуют так называемые *зомби-куки*, которые невозможно удалить. Все это сильно вредит репутации файлов куки, особенно в тех случаях, когда вредоносные куки обходят любые средства контроля и позволяют пользователям управлять обычными файлами куки.

Должен ли HTTP/2 решить проблему HTTP-куки?

Несмотря на то что в аспектах конфиденциальности и безопасности у пользователей сложилось негативное отношение к файлам куки, не было предложено никакой альтернативы. По свой сути, эти файлы не являются вредоносными и опасными; однако таковым может быть их использование. Альтернативы, которые позволяют добавить в HTTP состояния, например параметры URL и локальное хранилище[5], имеют аналогичные недостатки и дают пользователям меньше контроля над конфиденциальностью и безопасностью.

Некоторые люди считают, что HTTP/2 должен был решить эту проблему с помощью введения концепции состояний, реализовав тем самым более безопасное решение, которое не будет нарушать конфиденциаль-

---

1 https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis#section-4.1.3.

2 https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

3 https://tools.ietf.org/html/draft-ietf-httpbis-rfc6265bis#section-5.3.7.

4 https://nakedsecurity.sophos.com/2018/03/20/apple-burns-the-hsts-super-cookie/.

5 https://dev.to/rdegges/please-stop-using-local-storage-1i04.

ность. Однако, несмотря на частые попытки решить эту проблему[1], придумать что-то лучше файлов куки еще никому не удалось. Любое решение так или иначе должно также поддерживать традиционные файлы куки HTTP, иначе возникает риск того, что веб-сайты просто не будут поддерживать его. В HTTP нет концепции состояния, и даже несмотря на то, что в HTTP/2 она была добавлена на сетевом уровне (например, состояния потока и состояние динамической таблицы HPACK), на уровне приложения эта концепция по-прежнему не реализована. HTTP-сообщение, отправленное по одному соединению HTTP/2, может немного отличаться от сообщения, отправленного по другому соединению HTTP/2 на двоичном уровне (например, из-за сжатия заголовка), однако оно все еще остается тем же самым HTTP-сообщением без сохранения состояния для вышестоящего уровня.

Как уже было сказано в этой главе, в концепцию файлов куки были внесены некоторые нововведения. Вполне возможно, что в HTTP/2 могли бы обеспечить их полное соблюдение, вместо того чтобы делать их необязательными, как в HTTP/1. Однако в таком случае возникли бы проблемы с их принятием, поэтому разработчики HTTP/2 решили сохранить семантику HTTP/1 и внести изменения только на транспортный уровень.

Возможно, в будущем они захотят реализовать нечто лучшее, чем куки. Таким образом, не стоит отказываться от следующей версии HTTP, даже если в текущей версии решены еще не все проблемы.

10.1.3 HTTP и шифрование

Как и состояние, шифрование изначально не входило в принципы разработки HTTP; эта концепция была добавлена постфактум с HTTPS. HTTPS зашифровывает обычные HTTP-сообщения перед отправкой и разворачивает их после их получения, и в большинстве случаев это работает. Ранее высказывались опасения относительно стоимости сертификатов SSL/TLS и производительности шифрования и дешифрования, однако первое решалось с помощью дешевых (или даже бесплатных) сертификатов[2], а второе становилось лучше по мере увеличения вычислительной мощности[3]. Однако установка первых соединений по-прежнему влечет за собой снижение производительности, как описано в главе 9.

Нужно ли шифровать весь HTTP-трафик?

Большинство вопросов относительно HTTPS возникает на почве сложности начальной настройки и управления, зависимости от сторонних центров сертификации (ЦС)[4], а также многих беспокоит факт того, что использование HTTPS не является повсеместным. Последний аспект

[1] Например: https://github.com/mikewest/http-state-tokens.

[2] https://letsencrypt.org/.

[3] https://istlsfastyet.com/.

[4] https://www.howtogeek.com/182425/5-serious-problems-with-https-and-ssl-security-on-the-web/.

стал яблоком раздора после стандартизации HTTP/2. Многие считали, что следующая версия HTTP должна быть доступна только в защищенном формате (HTTPS) и что незащищенные HTTP-соединения должны остаться в прошлом. Другие были убеждены, во многих случаях шифрование не требуется, и, следовательно, оно не должно быть обязательным в протоколе. По иронии судьбы ко второй группе обычно относились те, кто сетовал, что HTTP/2 недостаточно улучшил конфиденциальность из-за файлов куки.

Вероятностное шифрование

Альтернативой шифрованию HTTPS, которое требует как шифрования, так и аутентификации (с использованием сторонних центров сертификации), является *вероятностное шифрование* для URL-адресов HTTP. Данный метод шифрует данные на транспортном уровне, но не требует подтверждения, что пользователи обращаются к подлинному веб-сайту. Безусловно, вероятностное шифрование – это шаг вперед в сравнении с HTTP, но шаг назад в сравнении с HTTPS. Такой тип шифрования можно без особых усилий развернуть на уровне протокола, не прибегая к обращению в сторонние центры сертификации.

HTTP/2 и шифрование

После длительных споров консенсус так и не был достигнут. В результате HTTP/2 был представлен с возможностью использовать как зашифрованное (h2), так и незашифрованное (h2c) соединение. Функция вероятностного шифрования не была реализована.

На практике все сложилось следующим образом: все веб-браузеры (основные клиенты HTTP) решили реализовать HTTP/2 через HTTPS (h2), как обсуждалось в главе 3. Так произошло как по идеологическим причинам (ведущие производители браузеров заявили о своем намерении перейти на зашифрованный Internet[1]), так и по техническим (в зашифрованных сеансах новый протокол может быть введен без использования соответствующей сетевой инфраструктуры промежуточного программного обеспечения). Microsoft был единственным поставщиком браузеров, который выразил заинтересованность в разрешении использования незашифрованного HTTP/2, однако в конце концов свет увидела только зашифрованная версия; по всей видимости, в Microsoft пришли к этому решению после обнаружения проблем с совместимостью, когда HTTP/2 не использовался поверх HTTPS.

Firefox позволяет использовать заголовок `Alt-Svc` для загрузки веб-сайтов через альтернативные службы, благодаря чему теперь некоторые поставщики предлагают HTTP/2 для HTTP-сайтов[2]. Такой метод работает

---

1 https://blog.mozilla.org/security/2015/04/30/deprecating-non-secure-http/ и https://blog.cloudflare.com/opportunistic-encryption-bringing-http-2-to-the-unencrypted-web/.

2 https://blog.cloudflare.com/opportunistic-encryption-bringing-http-2-to-the-unencrypted-web/.

для сайтов, имеющих версию HTTPS, но не готовых к переходу на нее (например, во избежание предупреждений о смешанном содержании). В Firefox заголовок `Alt-Svc` позволяет получать сайт по HTTPS (и HTTP/2), а представлять его как сайт HTTP; в каком-то смысле это представляет собой вероятностное шифрование «с изюминкой».

Возможно, требования относительно доступности HTTP/2 только через HTTPS могли быть преждевременными. За последние несколько лет использование HTTPS значительно возросло, но, к сожалению, до сих пор не приняло повсеместный характер. В главе 6 я говорил, что Firefox обслуживает более 70 % веб-трафика по HTTPS, однако сейчас статистика изменилась. Если смотреть на картину в целом, цифры менее оптимистичны. Топ-10 самых популярных сайтов обслуживает через HTTPS лишь 40 % веб-трафика[1] (см. рис. 10.3); однако эта цифра в перспективе будет расти. В 2015 году, после того как HTTP/2 был одобрен, по умолчанию HTTPS использовали только около 5 % сайтов.

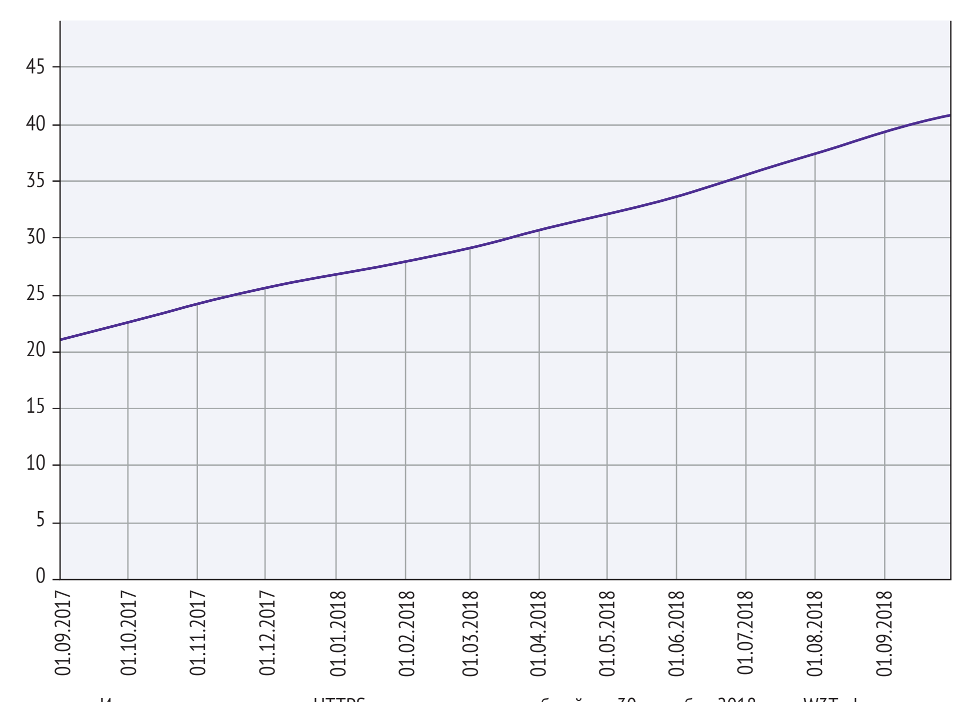


Рис. 10.3 Использование HTTPS топ-10 популярных сайтов

Таким образом, можно предположить, что для крупных сайтов более важны безопасность (и, раз уж на то пошло, производительность). Однако до момента, когда использование HTTPS станет обязательным, нам предстоит пройти еще долгий путь. Возможно, обязательный характер

---

1 https://w3techs.com/technologies/details/ce-httpsdefault/all/all.

HTTPS для HTTP/2 при стандартизации не был введен именно поэтому. Между стимулом для поощрения принятия хороших стандартов и появлением еще одного препятствия для принятия существует очень тонкая грань.

Также следует помнить и о существовании *внутреннего трафика*, не связанного с внешним Internet. Сюда входят сайты внутренней сети и *серверы внутренних приложений*, обслуживаемые веб-сервером, на котором выполняется внешняя реализация HTTPS. Внутренние веб-сайты, наряду с внешними, должны быть защищены с помощью HTTPS, однако это можно реализовать далеко не для всех инфраструктур. Сегодня Internet находится только на стадии перехода на шифрование отчасти по причине его открытого характера, ввиду которого возникают некоторые риски. Закрытая внутренняя сеть сопряжена с меньшими рисками, поэтому шифрование таких сайтов зачастую оставляют на потом. Внедрение HTTPS во внутренние сети отстает от внедрения HTTPS во внешнюю сеть; особенно это касается внутренних серверов, когда для отработки HTTPS-соединений используется пользовательский интерфейсный веб-сервер.

Если сайты, предназначенные только для внутреннего использования, не открывают доступ к Internet или не используют групповой сертификат (что дороже и не так легко автоматизировать), они не могут использовать коммерческие центры сертификации (особенно автоматизированные бесплатные центры сертификации, такие как Let's Encrypt). Многие прибегают к запуску внутренних центров сертификации, однако их крайне сложно настроить на автоматический выпуск и продление сертификатов.

HTTP/2 не является обязательным требованием для веб-сайтов; его можно рассматривать как вариант для тех, кто хочет увеличить скорость веб-сайта, а такие пользователи, скорее всего, в любом случае будут использовать HTTPS. Во внутренних сайтах задержки не так страшны, как во внешних, поэтому здесь преимуществ перехода на HTTP/2 становится несколько меньше. В предыдущих главах я уже говорил, что сегодня нет необходимости поддерживать HTTP/2 на внутренних серверах, если через HTTP/2 работает пограничный сервер, к которому подключаются пользователи. Тем не менее нет смысла изобретать новую версию основного протокола, если большая часть пользователей не сможет ей пользоваться. На момент написания этой книги эта часть составляет 60 % внешнего Internet, а сразу после стандартизации HTTP/2 она составляла 95 %. В частных внутренних сетях это число, вероятно, еще выше.

Помимо прочего, HTTP ограничен для распространения за пределы веб-сайтов и в доменах *интернета вещей* (Intrenet of Things, IoT), где управление сертификатами и HTTPS, безусловно, намного сложнее. Даже на момент написания этой книги в достаточной степени этот аспект не изучен. Однако, несмотря на это, были выдвинуты предложения относительно реализации HTTPS в *локальной сети*[1]. Устройства IoT

[1] https://www.w3.org/community/httpslocal/.

могут не нуждаться в HTTP/2, однако оставаться на HTTP/1 – сравнимо с шагом назад.

В конечном итоге не удивительно, что HTTP/2 не был стандартизирован с требованием HTTPS. Если тогда было слишком рано, то сейчас есть веские причины стремиться к этому. В протокол QUIC заложено согласование параметров HTTPS, что может иметь смысл, поскольку разница между протоколами составляет 3-4 года. Однако многие аргументы в пользу реализаций HTTP, не предназначенного для публичного доступа, по-прежнему остаются, и к адекватному решению общественность еще не пришла.

10.1.4 Проблемы транспортного протокола

Также одна из основных претензий касалось самого протокола на транспортном уровне. Если до этого HTTP не функционировал на этом уровне, за исключением отправки сообщений о том, что это был поток данных, которому предшествовала строка запроса (или ответа) и HTTP-заголовков, то в HTTP/2 ситуация в корне изменилась благодаря добавлению уровня двоичного фрейминга.

Нарушения иерархических представлений

Как уже говорилось в главе 1, сетевые протоколы часто построены на отдельных уровнях. HTTP/2 вышел за рамки своего уровня и приобрел некоторые характеристики TCP. На рис. 10.4 представлен веб-стек и то, как он (примерно) отображается на некоторых уровнях сетевой модели OSI (и тут есть некоторые совпадения). Уровень 6, например, не сопоставляется напрямую ни с одной из представленных технологий, хотя HTTP позволяет передавать информацию о форматах файлов.

HTTP/2 выходит за пределы уровня приложения, на котором так и остался HTTP/1. HTTP/2 управляет большим количеством элементов, традиционно считающимися прерогативой транспортного уровня (например, мультиплексирование и управление потоком). Поскольку HTTP/2 повторяет, а не заменяет обработку этих концепций в TCP, он как бы дополняет то, что традиционно считается HTTP. Таким образом, новый уровень двоичного фрейминга охватывает несколько уровней, как показано на рис. 10.5.

Многие люди считали «нарушение иерархических представлений» не лучшей идеей, ведь иерархия уровней протокола позволила упростить реализацию на каждом из уровней. На практике уровни не имеют между собой столь четких границ, и пользователям нежелательно зацикливаться на этом[1], поэтому предыдущий аргумент действительно может иметь смысл. В частности, повторение концепций TCP (хотя и с ограничениями), может привести к возникновению некоторых проблем, как обсуждалось в главе 9.

1 https://tools.ietf.org/html/rfc3439#section-3.

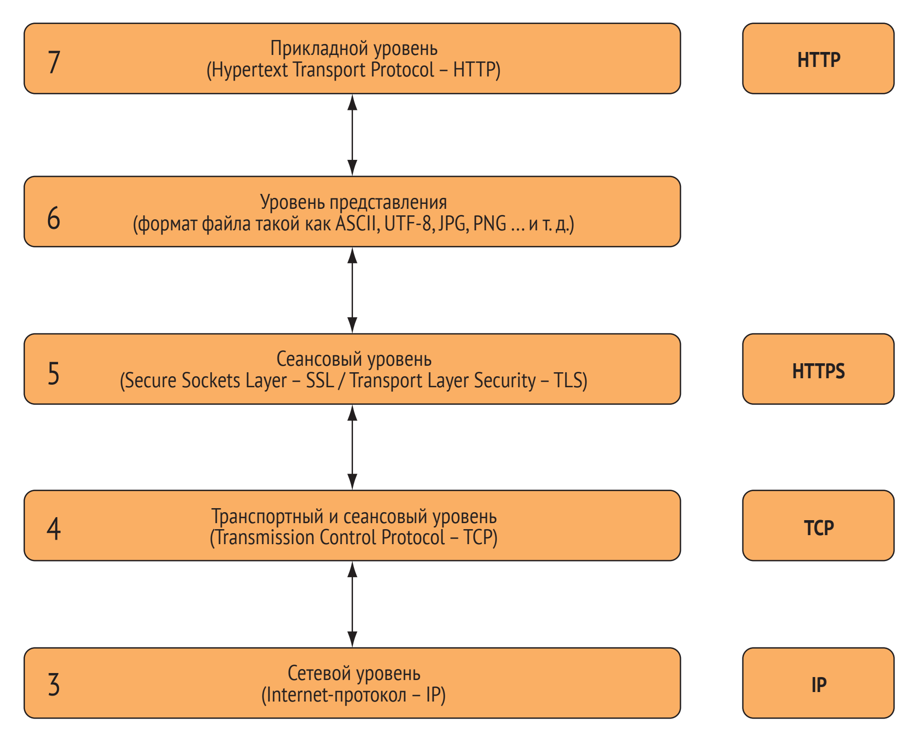


Рис. 10.4 Модель OSI и стек веб-сетей

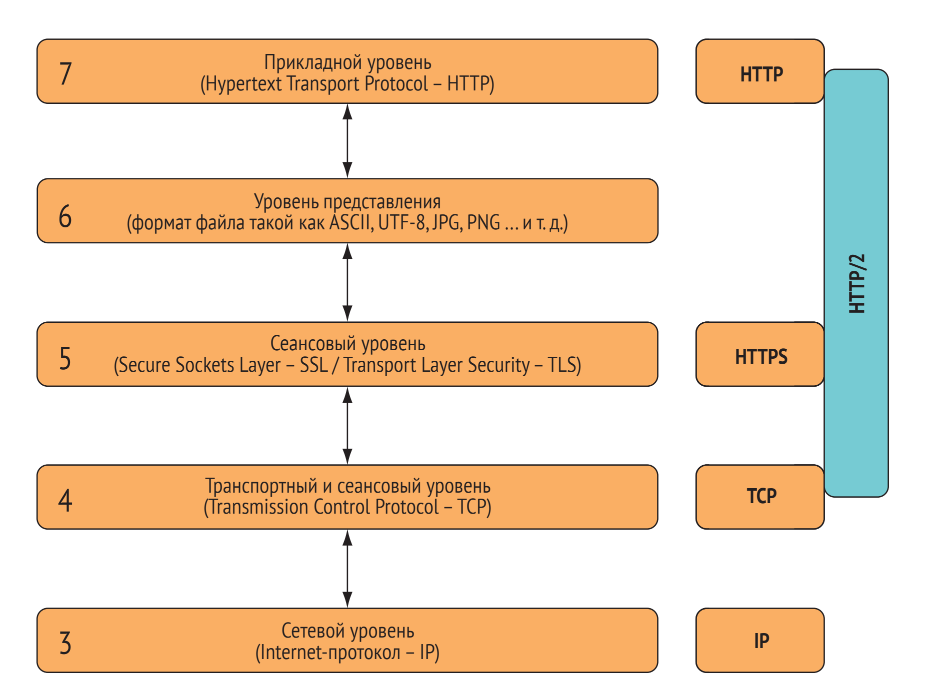


Рис. 10.5 Модель OSI и сетевой сетевой стек HTTP/2

QUIC нацелен на то, чтобы вернуть границу между транспортным протоколом приложения (HTTP) и сетевым транспортным протоколом (TCP и UDP), как показано на рис. 10.6. В этой модели HTTP/3 сопоставляет типы фреймов HTTP и показывает, как они должны использоваться для HTTP-трафика, но оставляет информацию транспортного уровня (например, управление мультиплексными потоками) для QUIC. Однако с другой стороны может показаться что, объединяя части уровней TCP и HTTPS, QUIC еще больше размывает границы между ними; но на самом деле это имеет смысл. HTTPS всегда был отделен от HTTP, поэтому он больше относится к сессионному и транспортному уровням, чем к уровню приложения.

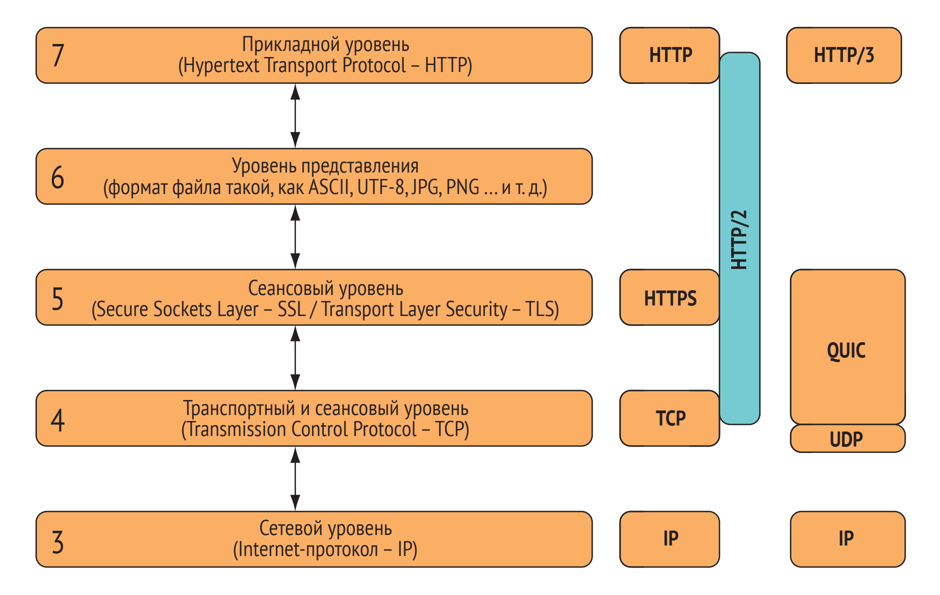


Рис. 10.6 Модель OSI и сетевой сетевой стек QUIC и HTTP/3

HOL-блокировка в TCP

В главе 9 я рассказывал о том, как мультиплексирование HTTP/2 решает проблему блокировки заголовка HTTP (HOL) и переносит ее на уровень TCP. Следовательно, в определенных условиях (при потерях) HTTP/2, возможно, работает медленнее, чем HTTP/1. На эту проблему уже давно обратили внимание. Она упоминалась в первом проекте HTTP/2[1] (в отличие от более поздних проектов), где говорилось, что преимущества использования одного TCP-соединения перевешивают отрицательные эффекты:

> *Использование нескольких соединений имеет свои плюсы. Например, SPDY мультиплексирует несколько независимых потоков в один, что создает потенциальную возможность возникнове-*

[1] https://tools.ietf.org/html/draft-ietf-httpbis-http2-00#section-4.3.

> *ния проблем с блокировкой заголовка на транспортном уровне. Согласно результатам проведенных тестов отрицательные эффекты блокировки заголовка (особенно при потере пакетов) перевешиваются преимуществами сжатия и приоритизации.*
>
> – Спецификация HTTP/2, версия 00

Согласно главе 9, QUIC стремится решить эту проблему, однако для большинства соединений подходит и HTTP/2. Стремление к разработке идеального решения привело бы к ненужной задержке разработки полезного обновления, не говоря уже о том, что переход с HTTP 1.1 на QUIC и HTTP/3 сам по себе чрезвычайно грандиозен.

Внедрение в HTTP концепцию состояния

Также в базовый протокол HTTP впервые была заложена концепция состояния. Файлы куки HTTP, о которых я рассказывал ранее, позволяют передавать состояние по HTTP, однако не добавляют состояние в сам протокол. Концепция состояния вводятся благодаря мультиплексированию потоков и внедрению их идентификаторов (глава 4), машине состояний HTTP/2 (глава 7) и сжатию заголовка HPACK (глава 8). Однако одно состояние все же не удалось ввести: это состояние сеанса, о котором я рассказывал в разделе 10.1.2. Данный выбор был осознанным, и он не так противоречив, как кажется. Несмотря на то что состояние было заложено в HTTP/2 на уровне подключения, на общем уровне HTTP это не имеет никакого значения. Тот же HTTP-запрос по-прежнему может быть выполнен через отдельное HTTP/2-соединение либо в параллельном соединении или вообще намного позже, и он по-прежнему будет передавать то же семантическое значение на уровне HTTP. Концепция состояний была введена исключительно для облегчения обработки мультиплексированного протокола и сжатия заголовка. Если состояние каким-либо образом собьется, соединение может быть разорвано и восстановлено, а HTTP-запрос может быть выполнен через новое соединение с теми же проблемами и ограничениями, что и в HTTP (например, когда повторная отправка безопасна [индемпотентна])[1]. Части HTTP/2 с отслеживанием состояния не важны ровно так же, как те же части TCP. Фактически состояние было добавлено только для того, чтобы работа потоков HTTP/2 больше походила на TCP.

Инструментальная поддержка HTTP/2

Бинарное разделение на слои и отказ от простого текстового формата раздражали многих людей, поскольку перестали работать такие простые инструменты, как Telnet (см. главу 1). Однако по большей части эти недовольства беспочвенны, поскольку зашифрованные HTTPS-соединения страдают теми же проблемами, и поэтому HTTPS получает соответствующую инструментальную поддержку. Возможно, более серьезным аргу-

[1] https://tools.ietf.org/html/rfc7231#section-4.2.2.

ментом было бы то, что некоторым сетевым устройствам, поддерживающим HTTP (например, HTTP-кеши, такие как Varnish) теперь приходится реализовать большее количество HTTP-трафика. Однако, даже несмотря на это, не стоит отказываться от развития протокола, и запрещать инфраструктуре маршрутизировать HTTP-трафик без полного понимания деталей HTTP-сообщения.

Действительно ли HTTP/2 является улучшением HTTP?

Таким образом, мы приходим к выводу, что большинство изменений в HTTP/2 коснулось транспортного уровня, а не уровня HTTP. Достаточно ли улучшений на уровне HTTP получила новая версия? А может быть, вместо HTTP/2 ее стоило назвать HTTP/1.2, TCP/2 или HTTPS/2? Дальнейшее развитие (а именно разработка QUIC) подтверждает, что многоуровневое распределение протокола не может быть рассмотрено как часть HTTP. В любом случае на эти вопросы мы никогда не найдем ответ. Новый протокол содержит критические изменения, поэтому он по определению отличается от предыдущей версии. Кроме того, разработчики редко присваивают обновлениям номера версий. Таким образом, переход спустя 20 лет на версию 2, которая включает в себя множество критических изменений и не имеет обратной совместимости, – это не самый плохой вариант.

10.1.5 HTTP/2 слишком сложен

Еще один предмет споров – это сложность HTTP/2. Нет сомнений в том, что это так, особенно по сравнению с простым на первый взгляд HTTP/1.1. Уровень фрейминга, его двоичная природа и внедрение концепции состояния – это действительно очень сложно (особенно в сравнении с HTTP/1.1), не говоря уже о приоритизации и сжатии заголовков. Чтобы разобраться в этих концепциях, вам нужна целая книга! Благодаря этому у авторов технической литературы появилось широкое поле для творчества, однако для протокола – это серьезная проблема. Одна из основных причин успеха HTTP – его простота. Главные детали HTTP умещаются на одну страницу спецификации (если так можно выразиться) HTTP/0.9; несмотря на добавленные версии HTTP/1.0 и 1.1, концептуально HTTP все еще легко понять.

Однако перед вами сейчас складывается неполная картина. HTTP/1.1 был прост для людей, но его было сложно реализовать в программном обеспечении. Для этого неформатированного, неструктурированного текстового протокола нужна дополнительная проработка всех возможных крайних случаев. В свою очередь HTTP/2 кажется намного более сложным, но все эти сложности легко поддаются автоматизации. Конечно, это не гарантирует, что все его реализации будут идеальны. Разработчикам все еще будет над чем поработать (еще одна причина для написания такой книги), однако стоит помнить, что такая ситуация может возникнуть при внедрении любой новой технологии, и особенно это от-

носится к HTTP/1. Сложность HTTP/2, как и HTTP/1, касается в основном разработчиков низких уровней, таких как разработчики веб-браузеров и веб-серверов, и почти не затрагивает веб-разработчиков более высокого уровня; и большинство разработчиков низкого уровня утверждало, что HTTP/2 реализовать проще, чем HTTP/1.

Возможно, количество реализаций является главным показателем степени сложности. На домашней странице веб-сайта HTTP/2 представлено более 80 отдельных активных реализаций[1]. HTTP/2 поддерживает большинство популярных веб-серверов и веб-браузеров; и многие из них запустили поддержку сразу после выхода новой версии. Безусловно, во всех этих реализациях были свои ошибки и проблемы, некоторые из них еще не устранены. Однако то, как быстро они приняли новую версию, вероятно, указывает на то, что эти сложности не представляют большой опасности. Впрочем, обратите внимание, что некоторые части спецификации (например, HTTP/2 push и определение приоритетов) ввиду их сложности заложены не во все реализации.

Трудно спорить с тем, что концепция HTTP/2 действительно сложна; однако это не такая серьезная проблема, как вы можете подумать. В большинстве случаев простота должна быть приоритетней сложности (принцип Keep It Simple Stupid [KISS]), однако, например, HTTP/1 не так прост, как кажется. Несмотря на все аргументы в пользу того, что сложность не имеет значения, те, кто обнаруживают проблемы с HTTP/2 в своих реализациях, на решение которых у них уйдут дни, несомненно, будут проклинать эту сложность. Я думаю, что все, кто сталкивались с таким, поймут насколько это неприятная ситуация.

10.1.6 HTTP/2 – временная мера

HTTP/2 не был нацелен на решение абсолютно всех проблемы HTTP. После нескольких лет застоя, имея на вооружении проверенный на деле SPDY, группа HTTP-WG IETF стремилась не увязнуть в кроличьей норе, поэтому многие проблемы не были решены полностью, а HTTP/2 был одобрен, несмотря на отсутствие единого мнения касаемо некоторых вопросов. Несомненно, эта ситуация разочаровала тех, кто считал, что в HTTP/2 многие проблемы могли бы быть решены. Однако подход разработчиков выглядит весьма прагматичным.

HTTP/2 улучшил производительность протокола, устранив некоторые из основных узких мест, поэтому его было удобно запускать и использовать на практике. Ничто не мешает будущей версии HTTP улучшить то, что не успели сделать в этой версии. Сегодня возникает множество предложений относительно улучшения протокола с помощью новых настроек и типов фреймов.

QUIC стремится решить некоторые проблемы, которые не удалось решить в HTTP/2. К ним относится HOL-блокировка TCP, обеспечение более полного шифрования, улучшенные процесса установки соедине-

---

[1] https://github.com/http2/http2-spec/wiki/Implementations.

ния и внедрение концепции миграции соединения. На появление этих изменений в HTTP/2 могло потребоваться еще четыре года, но, по всей видимости, в этом не было необходимости. Реализация QUIC, вероятно, займет больше времени, потому что он сложнее HTTP/2. На аналогичном этапе процесса стандартизации у QUIC было меньше реализаций, чем у HTTP/2, вероятно, из-за того, что реализации SPDY на HTTP/2 были намного проще. Однако gQUIC воспринимается несколько иначе. Для Internet, который почти полностью основан на TCP, переход на UDP будет сопряжен с массой проблем. Учитывая эту сложность, тот факт, что HTTP/2 является временным препятствием и в нем наблюдается некоторый прогресс, является скорее положительным, чем отрицательным.

10.2 HTTP/2 в реальном мире

Все эти аргументы были выдвинуты до официальной стандартизации HTTP/2, и ни один из них не рассматривался как причина отменить или отложить окончательную стандартизацию. С момента выпуска HTTP/2 и на момент написания этой книги HTTP/2 поддерживают более 30 % из 10 млн веб-сайтов[1] (рис. 10.7).

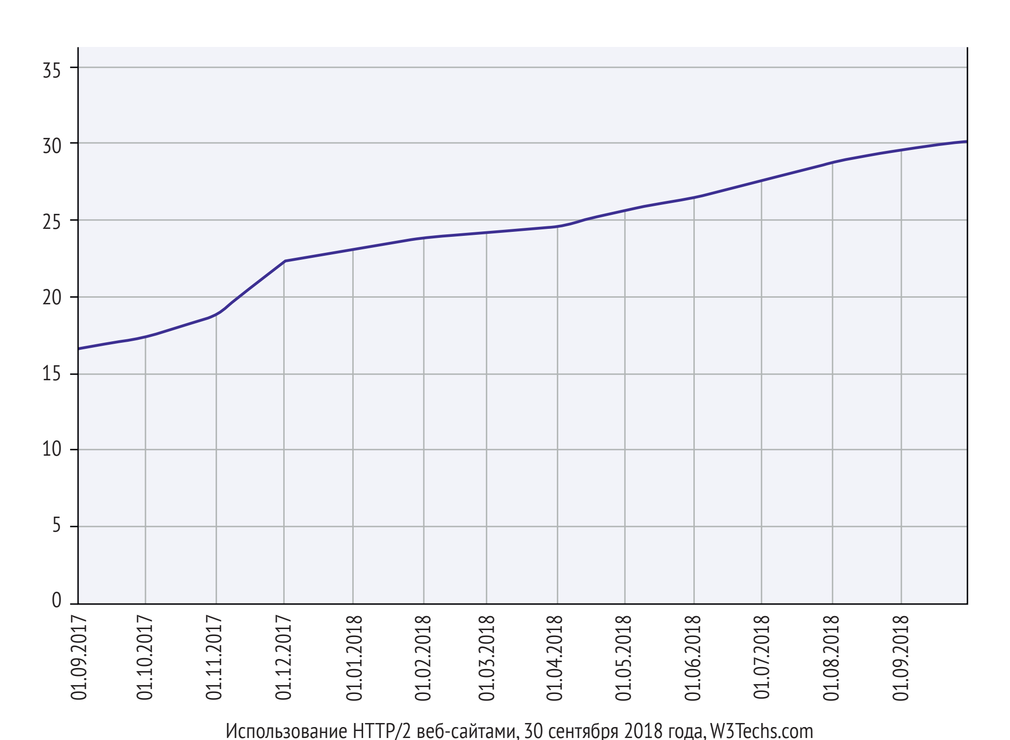


Рис. 10.7 Рост поддержки HTTP/2 с сентября 2017 года по сентябрь 2018-го

[1] https://w3techs.com/technologies/details/ce-http2/all/all.

Кроме того, на HTTP/2 основывается более 55 % веб-трафика[1], потому что крупные сайты, такие как Google, YouTube и Facebook, поддерживают именно эту версию и генерируют пропорционально больше трафика, чем небольшие сайты. Таким образом, согласно этой статистике, HTTP/2 уже зарекомендовал себя как успешный протокол.

Кроме того, в HTTP/2 уже был расширен и усовершенствован новыми настройками и типами фреймов, такими как *альтернативные службы*[2], фрейм `ORIGIN3` и *WebSockets*[4]. Существуют и другие предложения, направленные на дальнейшее расширение HTTP/2, в их число входит *Cache Digests*[5], а также многие другие инновации, находящиеся в стадии разработки.

В целом HTTP/2 стал частью Internet и продолжает расти как в плане использования, так и в плане функциональности. Все проблемы, обсуждаемые в этой главе, не лишены оснований, однако на практике они не являются критическими.

Безусловно, разработчикам удалось далеко не все. В частности, особого влияния пока еще не приобрела технология HTTP/2 push, в основном из-за сложностей правильного использования (см. главу 5), а отказ от поддержки на стороне сервера не помог исправить ситуацию. Возможно, оглядываясь назад, HTTP/2 push и не был панацеей для повышения производительности (как многие считали). Возможно, его следовало исключить из первоначального предложения HTTP/2 и добавить позже в качестве необязательного фрейма, использующегося при необходимости. Некоторые даже призывают исключить его из спецификации, хотя мне кажется, что такие люди делают поспешные выводы.

Кроме того, сложности при реализации HTTP/2 показывают, что перейти на него не так просто, как вам хотелось бы (что неудивительно для нового крупного протокола), и переход еще не гарантирует прироста производительности. У большинства сайтов все же наблюдается улучшение производительности, но бывают случаи, что производительность снижается. Чтобы получить максимальную отдачу от любой новой технологии, необходимо хорошо разобраться в ней. Я считаю, что HTTP/2 будет продолжать расти и развиваться, по крайней мере до тех пор, пока QUIC не будет оформлен и принят повсеместно, но ни один из них не заменит HTTP/1 во многих реализациях в ближайшее время, если вообще заменит.

---

[1] https://telemetry.mozilla.org/newpipeline/dist.html#!cumulative=0&measure=HTTP_RESPONSE_VERSION.

[2] https://tools.ietf.org/html/rfc7838.

[3] https://tools.ietf.org/html/rfc8336.

[4] https://tools.ietf.org/html/rfc8441.

[5] https://datatracker.ietf.org/doc/draft-ietf-httpbis-cache-digest/?include_text=1.

10.3 Будущие версии HTTP/2 и возможности HTTP/3 или HTTP/4

Теперь, когда набирает обороты HTTP/2, чего можно ожидать от следующей итерации HTTP? Будет ли это HTTP/2.1 или HTTP/3?

10.3.1 QUIC – это HTTP/3?

QUIC выводит концепции HTTP/2 на новый уровень, поэтому рассматривается как преемник, что по определению делает его HTTP/3. В июле 2018 года один из двух председателей рабочей группы HTTP заявил[1]: «Я рассматриваю QUIC как [HTTP/3] во многих отношениях… hq является логическим преемником h2», и в ноябре 2018 года рабочая группа HTTP согласилась, что HTTP-часть QUIC должна называться HTTP/3, и решила передать ее (и QPACK) рабочей группе HTTP после публикации[2]. Имя HTTP/3 не будет официально зарегистрировано до тех пор, пока QUIC не будет ратифицирован, поэтому нет гарантии, что новая версия будет называться именно так. Однако вероятность этого довольно высока.

QUIC позиционируется как более общий протокол транспортного уровня, однако выполняющий больше функций, чем HTTP, а HTTP/3 – это лишь один из вариантов использования QUIC. Во многих отношениях QUIC является преемником TCP, а не HTTP/2, но некоторые люди не согласны с обозначением TCP/2[3]. Еще и по этой причине, вероятнее всего, будет принято имя HTTP/3, что позволит различать более крупный протокол QUIC и часть HTTP. Разумеется, все это не означает, что функционирование и развитие HTTP/2 прекратится, точно так же после появления HTTP/2 пользователи не отказались от HTTP/1.1. Таким образом, многие реализации HTTP, скорее всего, останутся на TCP, используя HTTP/2 или HTTP/1.1. Однако, как только QUIC получит должное распространение, что займет некоторое время, HTTP/3 будет представлять лучшую версию HTTP и по идее должен использоваться везде, где это возможно (отсюда и название).

10.3.2 Дальнейшее развитие двоичного протокола HTTP

Если не брать во внимание различия и направленность версий QUIC (h3) и TCP (h2/h2c), то как следует расширить новый двоичный и мультиплексированный протокол в будущем?

Раньше HTTP-заголовки позволяли расширять основной протокол HTTP в зависимости от их использования клиентом и сервером. HTTP/2 добавляет новые возможности с новыми значениями настроек и новы-

---

1 Патрик Макманус (сопредседатель HTTP-WG), IETF 102 HTTPBIS, встреча II (https://youtu.be/tQAfDmW0qlI?t=588).

2 Рабочая группа HTTP IETF 103 (https://youtu.be/uVf_yyMfIPQ?t=4956).

3 https://github.com/HTTPWorkshop/workshop2016/blob/master/talks/quic.pdf.

ми типами фреймов. Значения настроек[1], в частности, позволяют обнаруживать новые возможности при установке соединения, что намного лучше, чем отправка HTTP-заголовка в надежде, что другая сторона поймет его.

Протокол был расширен фреймами `ALTSVC`, `ORIGIN` и `CACHE_DIGEST` (предлагаемый). Также были и другие предложения, такие как вторичные сертификаты[2], поэтому есть возможность расширения протокола благодаря новым типам фреймов[3]. В настоящее время нет острой потребности называть новую версию HTTP/2.1, поэтому рабочая группа HTTP отказалась от вспомогательного номера версии и назвала протокол HTTP/2, а не HTTP/2.0.

10.3.3 Развитие HTTP над транспортным уровнем

Если HTTP/2 и QUIC функционируют на нижнем транспортном уровне HTTP, то что происходит на более высоком уровне HTTP? Клиент и сервер управлялись устойчивым потоком новых заголовков HTTP, однако семантика HTTP не сильно изменилась со времен HTTP/1.1, в HTTP/2 этот уровень также остался нетронутым.

Как я упоминал ранее, одна из насущных проблем – это поиск альтернативы HTTP-файлам куки. Однако до сих пор еще не было ни одного подходящего предложения.

В других аспектах на высоком уровне HTTP оказался на удивление надежным, ввиду чего потребовались в первую очередь не расширения, а уточнения и настройки. Многочисленные расширения HTTP разрабатываются рабочей группой IETF[4], Web Platform Incubator Community Group (WICG)[5] или сторонними разработчиками, например такими как SPDY и QUIC, созданные Google. Большинство этих расширений можно реализовать, избежав фундаментального изменения протокола, вместо этого используя уже доступные методы расширения (заголовки HTTP, настройки HTTP/2 или новые типы фреймов HTTP/2).

Новые HTTP-методы

Особенно удивляет тот факт, что список основных HTTP-методов (`GET`, `POST`, `PUT`, `DELETE` и т. д.) не изменился со времен HTTP/1.1. Новые методы были представлены в Web Distributed Authoring and Versioning (WebDAV)[6] (такие как `PROPFIND`, `COPY` и `LOCK`), а также в некоторых RFC[7]. Последний до-

1 https://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml#settings.

2 https://tools.ietf.org/html/draft-ietf-httpbis-http2-secondary-certs.

3 https://www.iana.org/assignments/http2-parameters/http2-parameters.xhtml#frame-type.

4 https://github.com/httpwg/http-extensions.

5 https://www.w3.org/blog/2015/07/wicg/.

6 https://tools.ietf.org/html/rfc4918.

7 https://www.iana.org/assignments/http-methods/http-methods.xhtml.

бавленный метод был зарегистрирован в 2010 году (BIND). По большей части фундаментальными методами HTTP являются представленные 20 лет назад GET и POST, а также отчасти PUT и DELETE.

Добавить новые HTTP-методы довольно легко, однако в этом нет необходимости, поскольку большинство требований можно проксировать с помощью POST. В следующем примере переменная action позволяет передать необходимый метод, определяемый приложением (упорядочить указанный элемент):

```
:method: POST
:path: /api/doaction

{
    "action": "order",
    "item": 12345,
   ."quantity": 1
}
```

В других реализациях дополнительная информации (включая рекомендуемые действия) передается с помощью HTTP-заголовков:

```
:method: POST
:path: /api/doaction
action: order

{
    "item": 12345,
    "quantity": 1
}
```

Несмотря на то, что прошло уже 20 лет, я считаю, что в новых HTTP-методах сейчас нет необходимости, хотя в ближайшие годы, несомненно, они все же будут представлены. Я думаю, что они окажут серьезное влияние только на приложения, для которых они были предназначены.

Новые HTTP-заголовки

С течением времени возникали все новые и новые HTTP-заголовки, и я считаю, что их количество продолжит расти. HTTP/2 категорически запрещает использование новых полей псевдозаголовков[1], которые начинаются с двоеточия (:method, :scheme, :author, :path и :status). Однако эти поля могут добавляться новыми спецификациями (например, в HTTP/2 RFC вместе с загрузочными веб-сокетами был добавлен псевдозаголовок :protocol)[2].

Также заголовки могут добавлять и приложения. В спецификацию HTTP входят советы относительно новых полей заголовков[3], что говорит нам о том, что их список можно пополнять. Существует официальный ре-

1 https://httpwg.org/specs/rfc7540.html#PseudoHeaderFields.
2 https://tools.ietf.org/html/rfc8441.
3 https://tools.ietf.org/html/rfc7231#section-8.3.

естр заголовков сообщений[1] (включая те, которые используются HTTP), но многие приложения используют и незарегистрированные заголовки.

HTTP-заголовки позволяют реализовать новые функции независимо от их типа; это может быть и дополнительная информация для сторон (например, «Этот ответ использует формат XXX»), и подсказки (например, «К вашему сведению, я поддерживаю следующие форматы»), и информация для аутентификации (например, файлы куки), а также информация о маршрутизации («Переадресация была осуществлена с этого IP-адреса») и многое другое. Приложения с обеих сторон могут работать с заголовками, даже если промежуточная HTTP-инфраструктура (например, HTTP-сервер или веб-браузер) не может их передать.

Также последние несколько лет наблюдается тенденция к использованию HTTP-заголовков в целях безопасности. Зачастую такие заголовки отправляются с веб-сайта в браузер в ответах и содержат инструкции по включению функций безопасности, таких как Content-Security-Policy (CSP)[2] и HTTP Strict-Transport-Security (HSTS)[3]. Также с их помощью можно предоставить серверу дополнительную информацию о клиенте (веб-браузере или другом), например спецификацию Client Hints[4], которая должна позволять доставлять различный контент в зависимости от типов поддерживаемого клиентом содержимого.

Несомненно, по мере роста использования HTTP будет добавлено много новых заголовков функций на стороне клиента и сервера. Однако сейчас все это не нужно и не должно быть обязательным требованием в новых версиях HTTP.

Новые форматы

После того как в HTTP/1.0 появились заголовки, HTTP начал поддерживать самые разные форматы файлов и позволял использовать различную кодировку содержимого (например, методы сжатия, такие как gzip и br). Опять же, расширить HTTP в этом аспекте очень просто, и для этого даже не нужно создавать новую версию протокола под новым номером.

Новые коды состояния

Коды состояния[5] представляют собой еще один способ расширения функциональности HTTP. И для этого также не требуется разработка новой версии, по крайней мере, для тех кодов, которые будут близки к группам кодов, определенным основной спецификацией HTTP[6], как представлено в табл. 10.1.

---

1 https://www.iana.org/assignments/message-headers/message-headers.xhtml.
2 https://w3c.github.io/webappsec-csp/.
3 https://tools.ietf.org/html/rfc6797.
4 https://tools.ietf.org/html/draft-ietf-httpbis-client-hints.
5 https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml#http-status-codes-1.
6 https://tools.ietf.org/html/rfc7231#section-6.

Таблица 10.1 Группы кодов состояния HTTP

Код	Тип	Описание
1xx	Информационный	Запрос был получен; процесс продолжается
2xx	Свидетельствующий об успешном завершении	Запрос был успешно получен, понят и принят
3xx	Перенаправление	Для выполнения запроса необходимо предпринять дальнейшие действия
4xx	Ошибка клиента	Запрос содержит неверный синтаксис или не может быть выполнен
5xx	Ошибка сервера	Серверу не удалось выполнить действительный запрос

Например, код состояния HTTP `103`[1] был введен в конце 2017 года и не потребовал создания новой версии HTTP. Однако при его первом использовании во многих реализациях HTTP (например, в веб-браузерах) обнаружились проблемы. Реализации не ожидали получения более одного HTTP-ответа, так как до этого информационные ответы 1xx использовались нечасто (только при конкретных обстоятельствах). Технически для этого изменения не требовалась новая версия. Однако на практике этот код состояния функционировал несколько иначе, чем существующие, и для многих клиентов он стал критическим изменением до тех пор, пока в реализации не внесли доработки.

Помимо прочего, существует возможность добавлять новые категории кодов состояний (6xx, 7xx, 8xx, 9xx), вместо того чтобы расширять уже существующие группы. За 20 лет существования HTTP не было необходимости расширять категории кода состояния, однако в будущем это возможно.

10.3.4 Какие расширения требуют создания новой версии HTTP?

Учитывая наличие способов расширения HTTP, неудивительно, что версия HTTP за последние 20 лет не изменилась. Какие изменения потребуют создания новой версии? На данный момент этого никто не знает! Некоторые проблемы HTTP/2 (большинство из которых решается в рамках QUIC и HTTP/3) не были решены даже в результате мозгового штурма рабочей группа HTTP[2]. На сегодняшний день нет четкого представления о том, какие изменения может повлечь за собой создание совершенно новой версии (минуя HTTP/3).

Одно можно сказать наверняка: как и HTTP/2 и HTTP/3, следующая основная версия (для того чтобы получить новый номер) должна содержать критические изменения, не имеющие обратной совместимости. Будут ли это изменения в формате сети (как в HTTP/2), изменения на транспортном уровне (как в HTTP/3, переходящем на QUIC через UDP) или что-то еще, еще предстоит выяснить.

---

[1] https://tools.ietf.org/html/rfc8297.

[2] https://github.com/HTTPWorkshop/workshop2016/wiki/Future-of-HTTP#http3.

10.3.5 Как могут быть представлены будущие версии HTTP

Внедрение HTTP/2 предоставляет больше возможностей для расширения протокола, а также прокладывает дорогу для новой версии с использованием ALPN или других методов обновления, подробно описанных в главе 4. Если в какой-то момент потребуется критическое изменение, его внедрение должно стать проще. Таким образом, такая система должна привести к большему количеству инноваций, чем за последние 20 лет.

HTTP/3 стремится отойти от TCP, и в случае успеха в будущем это откроет нам путь для перехода к совершенно новым базовым технологиям. Будут ли разработчики использовать для доставки HTTP-сообщений что-то другое, кроме TCP и UDP? Изменится ли IP? Ответов на эти вопросы нет. Однако совершенно точно ясно одно: настала новая эра развития интернет-протоколов, и у разработчиков наконец-то есть методы для их внедрения.

10.4 HTTP как базовый транспортный уровень

А что насчет изменений в использовании HTTP? Что ждет протокол в этом аспекте? Первоначальным вариантом использования HTTP были веб-страницы, однако многие разработчики стремятся реализовать популярный HTTP за пределами этого уровня. HTTP – это довольно простой протокол, который поддерживается повсеместно, поэтому у него существует множество реализаций и библиотек. Для HTTP/1.1, по крайней мере, легко создать простой HTTP-сервер или клиент с любым программным обеспечением, которое позволяет читать или записывать TCP-канал (хотя, как обсуждалось ранее, текстовая природа HTTP/1.1 может привести к множеству проблем, которые изначально скрыты этой кажущейся простотой).

HTTP – это гибкий протокол, который готов к реализации вне своего изначального предназначения. HTTP обеспечивает связь между отдельными системами стандартизированным и понятным способом. От сложных приложений, использующих REST API или аналогичную технологию, до устройств интернета вещей HTTP используется в веб-приложениях и непосредственно вне веб-приложений.

Приложения могут использовать HTTP разными способами:

- использовать семантику и сообщения HTTP для доставки внутреннего трафика;
- использовать концепцию двоичного фрейминга HTTP/2;
- использовать HTTP для запуска другого протокола.

В оставшейся части главы мы подробнее рассмотрим эти варианты.

10.4.1 Использование семантики и сообщений HTTP для доставки внутреннего трафика

Данный метод является наиболее популярным методом использования HTTP за пределами веб-страниц. Сообщения API могут быть отправлены

через HTTP в любом формате, который использует клиент и сервер (XML, JSON или какой-либо проприетарный формат). Часто такие сообщения для взаимодействия между службами используют распространенные методы HTTP (GET, POST, PUT и DELETE). Например, архитектура микросервисов использует небольшие независимые службы, для которых HTTP – это отличный вариант. Другие протоколы также можно легко смоделировать с помощью HTTP (см. сноску ниже).

DNS через HTTPS (DoH)

Система доменных имен (Domain Name System, DNS) – это относительно простой протокол, который обычно включает в себя выборку одного типа записи из центрального каталога. Традиционно DNS был проприетарным протоколом на отдельном порту (порт 53), а также его можно было реализовать с помощью простого запроса GET.

Одной из причин перехода на HTTP стало использование шифрования, обеспечиваемого HTTPS. Раньше DNS был незашифрованной системой, и попытки внедрить шифрование породили более сложные протоколы, такие как DNSSEC и DANE, которые имеют свои недостатки, выходящие за рамки этой книги. HTTPS проверен, безопасен (при правильной настройке) и поддерживается повсеместно. Объединив интерфейс HTTPS с DNS, разработчики могут решить проблему шифрования DNS, о которой они спорят десятилетиями. Эта система известна как DNS over HTTPS[a], или DoH.

Поверх DNS вместо HTTPS разработчики могут использовать TLS, и для этой цели также существует отдельная спецификация[b]. Однако HTTPS проще реализовать, а также этот путь позволяет использовать и другие преимущества HTTP. Они включают широко поддерживаемое, удобное для прокси использование HTTP/2 или QUIC в мультиплексных запросах и использование HTTP/2 push для отправки нескольких ответов.

На момент написания этой книги на стороне сервера DoH поддерживают Google[c] и Cloudflare[d]; также поддержку добавил Firefox и даже начал экспериментировать с этим[e] и, кроме того, получил интересные результаты[f]. Они говорят нам о том, что HTTP является достаточно мощным инструментом для приложений, даже если у них уже есть собственный протокол передачи!

---

[a] https://tools.ietf.org/html/draft-ietf-doh-dns-over-https.

[b] https://tools.ietf.org/html/rfc7858.

[c] https://tools.ietf.org/html/rfc8484.

[d] https://developers.cloudflare.com/1.1.1.1/dns-over-https/.

[e] https://blog.nightly.mozilla.org/2018/06/01/improving-dns-privacy-in-firefox/.

[f] https://hacks.mozilla.org/2018/05/a-cartoon-intro-to-dns-over-https/.

[g] https://blog.nightly.mozilla.org/2018/08/28/firefox-nightly-secure-dns-experimental-results/.

IETF опубликовал спецификацию «Об использовании HTTP в качестве основы» (On the Use of HTTP as a Substrate)[1], в которой перечислены рекомендации и передовые методы, позволяющие получить максимальную отдачу от использования HTTP таким способом. Некоторые приложения расходятся со стандартами основных спецификациями HTTP и используют только часть протокола. Однако документ это допускает, но все же предупреждает, почему это может привести к потере некоторых преимуществ использования HTTP. HTTP можно использовать таким образом по многим причинам, некоторые из которых могут быть не очевидны, и в спецификации предпринимается попытка прояснить эти причины для пользователей протокола.

10.4.2 Использование концепции двоичного фрейминга HTTP/2

Появление уровня мультиплексированного двоичного фрейминга представляет новые возможности и причины для использования HTTP. В то время как HTTP предлагали отвергнуть ввиду его неэффективности в пользу прямых TCP-соединений или, возможно, WebSockets, HTTP/2 решает многие из этих проблем и становится его прямым соперником. Новый протокол gRPC от Google (сокращение от Google Remote Procedure Calls)[2] использует семантику HTTP и уровень двоичного фрейминга HTTP/2[3] для реализации более эффективного API на основе буферов[4] протокола вместо менее эффективных форматов, таких как JSON. Маловероятно, что Google не воспользовался бы преимуществами, предлагаемыми HTTP/2.

Многие пользователи могут захотеть применить концепцию двоичного фрейминга HTTP/2 для трафика, отличного от HTTP, в качестве полнодуплексного протокола, а не инициируемого клиентом протокола, как HTTP/2, только с ограниченным HTTP/2 push для запросов от сервера к клиенту. В настоящее время эта функция не поддерживается. Возможно, QUIC, созданный с нуля с учетом использования других протоколов, лучше подходит для этой цели (или, возможно, любые такие реализации будут перенесены в HTTP/2 через TCP). Сейчас разработчики могут использовать HTTP как способ запустить другой протокол, например полноценный двусторонний протокол.

10.4.3 Использование HTTP для запуска другого протокола

Еще один вариант нестандартного использования HTTP – использование его для начального обмена с целью перехода на другой протокол. HTTP идеально походит для этого, поскольку он поддерживается повсеместно.

---

1 https://tools.ietf.org/html/rfc3205 сейчас находится в процессе обновления, поэтому вскоре должен быть заменен.

2 https://grpc.io/faq/.

3 https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md.

4 https://developers.google.com/protocol-buffers/docs/overview.

Таким образом, другой протокол будет выглядеть как HTTP и поддерживаться так же широко, особенно через прокси и других посредников. Чтобы реализовать этот вариант, вы можете воспользоваться HTTP-методом `CONNECT` или обновить соединение (например, на WebSockets). В обоих случаях соединения будут устанавливаться как HTTP, а затем быстро перейдут на другой протокол. Однако для сторонних наблюдателей все это будет выглядеть как HTTP-трафик.

HTTP-метод CONNECT

Данный метод появился еще в версии HTTP/1.1. Он позволяет использовать HTTP-соединение в качестве прокси-сервера для туннелирования его к альтернативному серверу и порту. Метод часто использовался для применения этой функции к HTTPS. Делалось это посредством следующего синтаксиса:

```
CONNECT example.com:443 HTTP/1.1
```

Для HTTP/2 синтаксис будет таким:

```
:method: CONNECT
:authority: example.com:443
```

Данный код создает новое соединение от HTTP-сервера к example.com и позволяет сообщениям проходить от клиента к этому серверу, как показано на рис. 10.8.

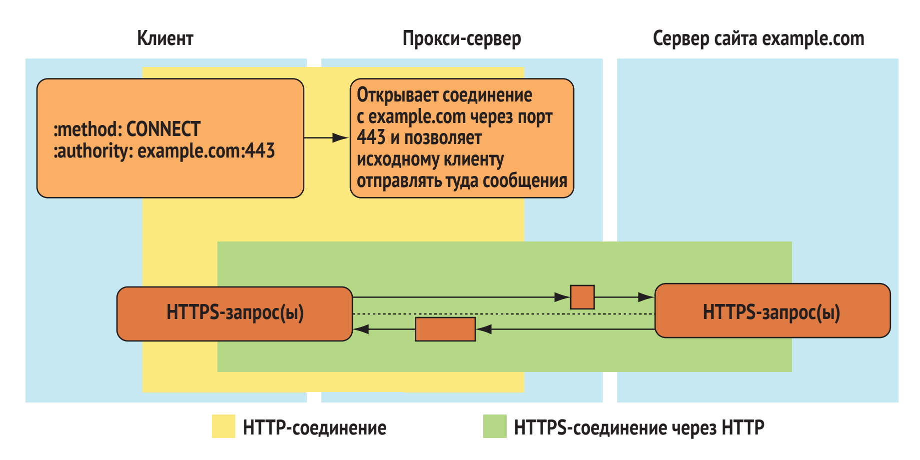


Рис. 10.8 Туннелирование HTTPS-соединения через HTTP-прокси-соединение с помощью CONNECT

Обратите внимание, что такой прокси-сервер отличается от прокси-сервера типа с функцией посредника, который создает два отдельных HTTP-соединения. В этом сценарии используется одно HTTPS-соединение и как минимум два TCP-соединения, поэтому после начальной настройки создается впечатление, что клиент подключается непосредственно к конечному серверу.

Прокси-серверы не получат доступа к HTTPS-соединению, поскольку оно будет сквозным. Таким образом, с помощью этой функции мы можем устанавливать HTTP/2-соединения через прокси-сервер HTTP/1.1, если клиент и сервер поддерживают HTTP/2 (см. рис. 10.9).

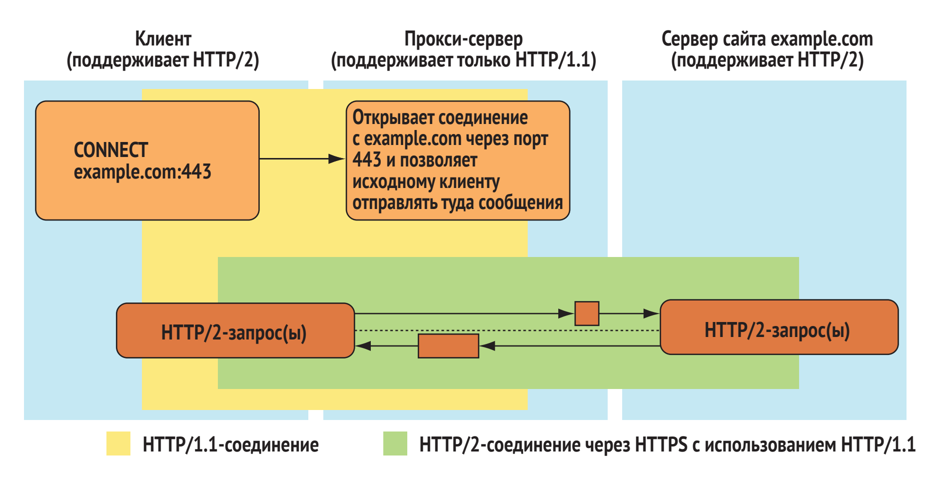


Рис. 10.9. Туннелирование HTTP/2-соединения через прокси HTTP/1.1 с помощью CONNECT

Туннелированное соединение, за исключением начального сообщения, не требует передачи HTTP-сообщений, ввиду чего может использовать любой протокол. На рис. 10.10 HTTP используется для подключения к example.com через порт 22 (используемый SSH). После этого любые сообщения, отправляемые по этому соединению, станут управляющими сигналами SSH, а не HTTP.

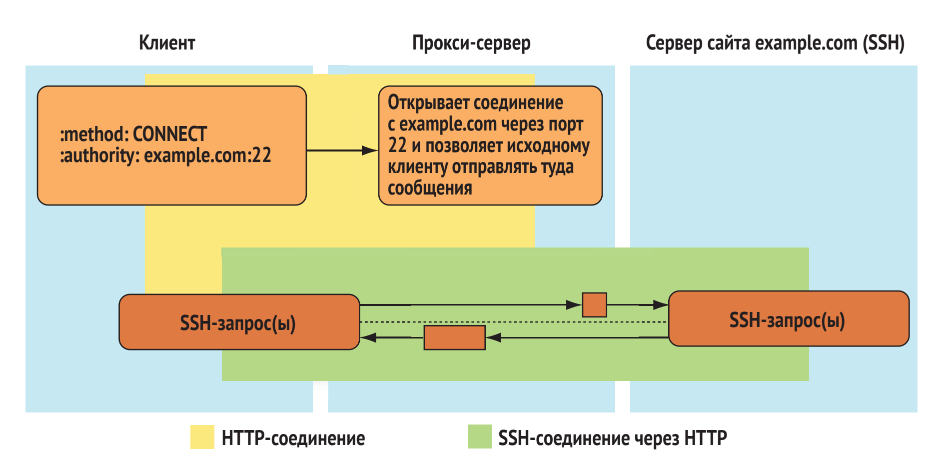


Рис. 10.10 Туннелирование SSH-соединения через HTTP-соединение с помощью CONNECT

Данная настройка может быть полезна для того, чтобы разрешить использование недоступных непосредственно из системы альтернативных протоколов (при условии, что они доступны на прокси-сервере). Все эти примеры показывают, что с помощью HTTP можно внедрить в какую-либо среду другие протоколы. Таким образом, вопрос об обработке этих протоколов сетевой инфраструктурой отпадает сам собой, особенно когда для сокрытия деталей от самих прокси-серверов используется HTTPS. Возможно, этот вариант даже интереснее, чем QUIC, перешедший от TCP к более простому протоколу UDP. На данный момент для поддержки метода `CONNECT` HTTP/3 обязательно потребуется TCP, однако эта проблема находится на стадии решения.[1] Для того чтобы ее решить, требуется настройка поддержки прокси-серверов как со стороны клиента, так и со стороны сервера.

Обновление HTTP-соединения (например, на WebSockets)

Еще один вариант – запустить HTTP-соединение, а затем обновить его до альтернативного протокола. Именно так работает WebSockets. Ниже вы можете увидеть такое рукопожатие с синтаксисом HTTP/1.1.

Запрос клиента (в целях упрощения другие поля заголовка WebSockets не представлены):

```
GET /application HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
```

Ответ сервера:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

Затем HTTP-соединение, которое все еще работает через TCP-порты 80 или 443, преобразуется в соединение WebSockets. Для небольших соединений WebSockets более эффективен, чем HTTP, поскольку отсутствуют накладные расходы для HTTP-заголовков (значительно уменьшенных в HTTP/2 благодаря HPACK, но все же затратных). Также WebSockets обеспечивает полнодуплексную связь. Поэтому использование этого протокола может быть полезно для приложений, обновляющихся в реальном времени, таких как чаты, финансовые приложения или спортивные новостные веб-сайты.

Данный метод нельзя реализовать в HTTP/2, поскольку он потребует обновления всего HTTP-соединения, что для мультиплексированного соединения, в котором должен обновляться только поток, бессмысленно. Следовательно, для HTTP/2 WebSockets должен использовать метод `CONNECT` [2] (с небольшими изменениями, позволяющими указать протокол).

---

1 https://tools.ietf.org/html/draft-pardue-httpbis-http-network-tunnelling.

2 https://tools.ietf.org/html/draft-ietf-httpbis-h2-websockets.

Запрос клиента (в целях упрощения некоторые поля заголовка WebSockets упущены):

```
:method = CONNECT
:protocol = websocket
:scheme = https
:path = /application
:authority = example.com
```

Ответ сервера:

```
:status = 200
```

После этих шагов в соединении данными WebSockets можно обмениваться в любом направлении. Другие потоки HTTP/2 в этом соединении могут продолжать использовать HTTP, другие соединения WebSockets или даже иной подобный протокол. WebSockets открывает множество возможностей, которые затрудняет природа HTTP (а именно то, что это протокол запроса и ответа). Тот же метод подходит для переключения на любой другой протокол, и, таким образом, я завершаю главу еще одним способом использования HTTP!

В последней главе я попытался заглянуть в будущее, чтобы увидеть, что ждет HTTP дальше. Сегодня относительно этого есть много предположений и крайне мало подтвержденной информации, поэтому на данном этапе никто не знает, как на самом деле будет выглядеть следующая версия HTTP. Однако вы можете быть уверены в двух вещах: HTTP останется важной частью Internet и его популярность будет расти. В начале главы 1 я уточнил, что термин «Всемирная паутина» часто ошибочно отождествляется с Internet как комплексом физических и логических протоколов. Однако непрерывное развитие Всемирной сети (а также специально созданного для нее протокола HTTP) может означать, что очень скоро такое смешение понятий окажется не столь далеким от истины, как раньше. В последней главе я объясняю различные варианты, которые позволяют расширить использование HTTP и будут способствовать его дальнейшему росту.

Читатели, дошедшие до этого момента, уже должны хорошо разбираться в HTTP/2 и связанных с ним технологиях. Также теперь вы знаете, что, хотя HTTP/2 и предлагает большое количество возможностей и методов для ускорения, он не гарантирует повышения производительности. Те, кто искал в этой книге быстрое решение, могут быть несколько недовольны, однако все же HTTP/2 работает быстрее для большинства сайтов; кроме того, не стоит забывать, что протокол находится в стадии развития и повсеместного принятия. QUIC и HTTP/3 будут использовать те же концепции, что и HTTP/2, однако выведут их на новый уровень. Знания, полученные благодаря этой книге, помогут вам ощутить все преимущества QUIC. Однако до этого вам еще предстоит многое узнать о HTTP/2 и о том, как его лучше всего использовать. Я надеюсь, что эта книга вызвала у вас интерес к HTTP/2. Я хочу, чтобы она стала отправной точкой для дальнейших экспериментов и наблюдения за развитием этого интересного, широко используемого протокола. Желаю удачи!

Резюме

- В процессе стандартизации вокруг HTTP/2 возникли некоторые разногласия и появилось немало недоброжелателей. Однако это не помешало внедрению HTTP/2 в Internet, и в большинстве случаев многие проблемы оказались не такими серьезными, как предполагалось.
- HTTP/2 получил широкое распространение и уже по этой причине может считаться успешным протоколом.
- HTTP теперь состоит из уровня двоичного фрейминга HTTP/2 и семантики HTTP, на которой строятся фреймы.
- QUIC обещает быть преемником HTTP/2, и когда он будет опубликован, HTTP-часть спецификации QUIC будет называться HTTP/3.
- Существует множество HTTP-методов для расширения вариантов использования протоколов, а благодаря HTTP/2 их становится еще больше.
- HTTP может использоваться для передачи других протоколов.
- HTTP ждет большое будущее!

Приложение
Обновление популярных веб-серверов до HTTP/2

В этом приложении описываются методы включения поддержки HTTP/2 на некоторых популярных веб-серверах напрямую (A1) или в качестве обратного прокси (A2). Обратите внимание, что предложение не является исчерпывающим справочным материалом для всех веб-серверов, поддерживающих HTTP/2; в первую очередь обратитесь к документации своего веб-браузера.

A.1 Обновление вашего веб-сервера для поддержки HTTP/2

В следующих разделах я расскажу, как установить совместимые с HTTP/2 версии обычных веб-серверов, а также как настроить их для поддержки HTTP/2. Обратите внимание, что этот список далеко не полный и со временем будет меняться; он призван лишь показать некоторые примеры настройки веб-серверов, совместимых с HTTP/2. Многие шаги в разделах повторяются, поэтому рекомендую сразу перейти к интересующему вас разделу.

Самозаверенные сертификаты HTTPS и ошибки сертификата

В большинстве примеров в этом приложении используются самозаверенные сертификаты SSL/TLS (также известные как сертификаты HTTPS). Такие сертификаты поставляются с веб-сервером или создаются с помощью команды `openssl` и т. п. Для того чтобы сертификат распознавался браузером и был надежным, он должен быть выпущен авторизованным центром сертифика-

ции (ЦС). Каждый браузер или операционная система содержит список центров спецификации. Фиктивные сертификаты, которые поставляются с веб-серверами (и те, которые представлены в этом приложении), не выдаются центром сертификации, и они известны как *самозаверенные сертификаты*. Они позволят сайту работать через HTTPS, но вы получите уведомление об ошибке сертификата. Обычно ошибки такого рода можно проигнорировать и легко получить доступ к сайту, однако это в некоторой степени опасно.

В случаях, если пользователь игнорирует ошибки сертификата, Chrome и Opera не используют кеширование HTTP по соображениям безопасности[a], поэтому нельзя использовать HTTP/2 push.

Можно добавить самозаверенный сертификат в хранилище доверенных сертификатов компьютера, чтобы браузер распознавал сертификат, отображал зеленый замок и устранял эти ошибки. Данный метод рекомендуется для локальной разработки (через localhost). Как это сделать, зависит от вашего браузера и операционной системы, но обычно действие выполняется двойным кликом по сертификату.

Если вы используете сервер с реальным доменным именем, стоит все же получить настоящий сертификат, тогда его будут распознавать все браузеры и не требовать при этом дополнительных шагов для его регистрации. Данная опция недоступна для localhost или любого другого локального IP-адреса (например, 127.0.0.1 или :: 0).

[a] https://bugs.chromium.org/p/chromium/issues/detail?id=103875#c8.

A.1.1 Apache

В версии 2.4.17 HTTP-сервер Apache (он же httpd) представил поддержку HTTP/2 через модуль `mod_http2` (именуемый ранее как отдельный модуль `mod_h2`[1]). Он считался экспериментальным вплоть до выхода версии 2.4.26. Я рекомендую использовать последнюю версию Apache (доступную на http://httpd.apache.org/), так как вышеупомянутый модуль получил в ней значительное развитие.

Apache поддерживает HTTP/2 через HTTPS с использованием ALPN и никогда не реализовывал старый метод NPN согласования SPDY/HTTP/2. Следовательно, Apache требует как минимум OpenSSL 1.0.2 (или эквивалент), чтобы включить поддержку HTTP/2 даже для браузеров, которые все еще поддерживают NPN. Apache разрешает HTTP/2 через HTTP-соединения с открытым текстом (известные как h2c), хотя эта функция имеет ограниченное использование, поскольку ни один браузер не поддерживает ее. Apache использует библиотеку nghttp2 в качестве основы своей функциональности HTTP/2 и требует версии 1.5.0 или более поздней для полной функциональности на момент написания книги.

HTTP/2 push поддерживается в Apache (см. главу 5). Также сервер поддерживает модуль `mod_proxy_http2`, который позволяет подключаться

[1] https://github.com/icing/mod_h2.

к серверным системам через HTTP/2, хотя этот модуль все еще считается экспериментальным. На вопрос «Нужно ли всегда использовать HTTP/2?» есть ответ в главе 3, где объясняется, что использование HTTP/2 на всем пути к внутреннему соединению чаще всего не является необходимым.

Apache на Windows

Хотя Apache на Windows в производственных системах, возможно, не так распространен, как на Linux, версия для Windows часто используется разработчиками. Компиляция Apache из исходного кода для Windows выходит за рамки этой книги. Но если вы хотите запустить Apache локально, чтобы поэкспериментировать с HTTP/2, вы можете использовать готовые версии Windows из различных источников (к сожалению, не из Apache). Популярные варианты для версий Windows – Apache Haus[1], Apache Lounge[2] и пакеты XAMPP. Все это вы можете найти на веб-сайте Apache по адресу http://httpd.apache.org/docs/current/platform/windows.html#down. Выбор подходящего вариант зависит от следующего:

- *версии Visual C++*. Возможно, вам придется установить Visual C++ Distributable (от Microsoft). Имейте в виду, что Microsoft в документации ссылается на эти файлы по-разному (по годам выхода или по версии). Для удобства я соединю все в один список:
  – Visual Studio 2008: VC++ 9;
  – Visual Studio 2010: VC++ 10;
  – Visual Studio 2012: VC++ 11;
  – Visual Studio 2013: VC++ 12;
  – Visual Studio 2015: VC++ 14;
  – Visual Studio 2017: VC++ 15.

  Вы можете просмотреть установленные вами версии в панели управления вашей системы Windows во вкладке **Установка и удаление программ**. Установить новую версию несложно, и я рекомендую выбирать самую последнюю;
- *архитектуры* – выбор между 64-битной и 32-битной (также известной как x86). Если вы используете 64-битную операционную систему, я рекомендую 64-битный Apache. Если вы не используете 64-разрядную операционную систему, задумайтесь о переходе на нее (в наши то дни!). Вы можете просмотреть архитектуру, щелкнув правой кнопкой мыши по **Мой компьютер** и выбрать **Свойства** в контекстном меню. В появившемся окне вы должны увидеть архитектуру (**Тип системы**);
- *версии OpenSSL*. Вам потребуется версия 1.0.2 или новее, но некоторые сайты предлагают сборки для версии 1.1.0 или новее. Я рекомендую использовать последнюю версию везде, где это возможно.

Установить пакеты Apache Haus (и другие) с поддержкой HTTP/2 можно следующим образом.

---

1 https://www.apachehaus.com/cgi-bin/download.plx.

2 https://www.apachelounge.com/download/.

1. Загрузите соответствующую версию на основе трех предыдущих вариантов и распакуйте папку в нужное место (например, C:\Program Files\Apache\Apache24).
2. Отредактируйте файл conf\httpd.conf, чтобы изменить следующую строку:

```
Define SRVROOT "/Apache24"
```

указав в ней расположение вашего сервера, например:

```
Define SRVROOT "C:\Program Files\Apache\Apache24"
```

Убедитесь, что в конце строки нет косой черты (`C:\Program Files\Apache\Apache24\`), и не забудьте заключить путь в кавычки, если он содержит пробел (например, `Program Files`).
3. Убедитесь, что модуль `mod_http2` активирован и в нем нет комментариев (значка # перед ним):

```
LoadModule http2_module modules/mod_http2.so
```

4. Сохраните изменения в httpd.conf. Обратите внимание, что, если он находится в папке Program Files, могут потребоваться права администратора.
5. Запустите Apache, желательно из командной строки, чтобы увидеть ошибки:

```
cd "c:\Program Files\Apache\Apache24\bin"
httpd.exe
```

Ниже приведены некоторые распространенные ошибки, которые могут помешать работе этой команды:

- сообщение об ошибке содержит предупреждение об отсутствии VCRUNTIME140.dll или аналогичного файла, что указывает на то, что вы не установили необходимый компонент Visual C++;
- файл журнала предназначен только для чтения, поэтому вы увидите сообщение о невозможности открыть журнал ошибок. Щелкните правой кнопкой мыши по папке **C:\Program Files\Apache\Apache24\Logs**, выберите **Свойства** в контекстном меню и снимите флажок **Только для чтения**, если он выбран;
- всплывающее окно брандмауэра Windows просит вас предоставить ему доступ к этому веб-серверу. Вам нужно ответить «да»;
- в сообщении об ошибке говорится, что у вас нет доступа к портам 80 или 443. Скорее всего, этот порт использует другая программа. Наиболее часто это World Wide Web Publishing Service (она же называется IIS), которую можно найти во вкладке **Службы** компьютера. (Остановите эту службу и установите для нее параметр запуска **Вручную** вместо **Автоматически**, если вы не используете IIS, Skype или другой веб-сервер.)

6. После запуска Apache проверьте http://localhost. Там вы должны увидеть страницу приветствия Apache Haus. Затем попробуйте

перейти на https://localhost, где вы получите ошибку сертификата, если используете фиктивный сертификат по умолчанию. Пропустите ошибку сертификата, и вы увидите, что ваша страница работает на HTTP/2, если вы откроете инструменты разработчика и добавите столбец **Protocol**, как показано на рис. A.1.

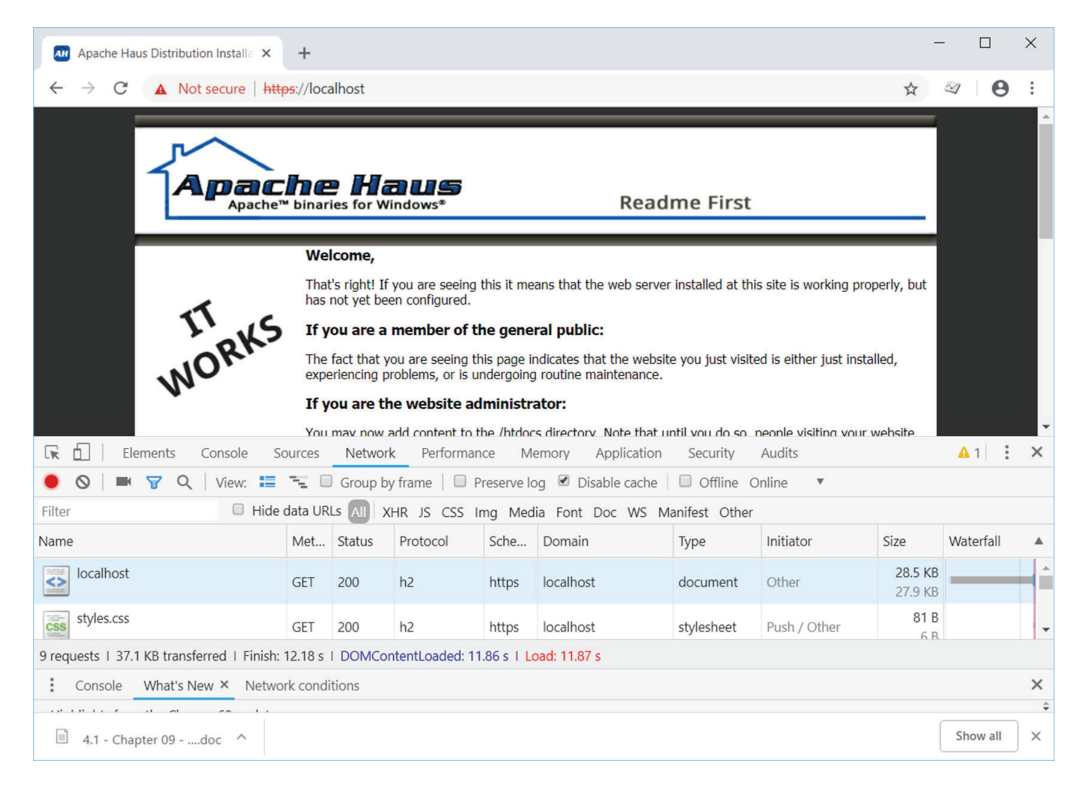


Рис. A.1 Запуск Apache на HTTP/2

7 Если Apache работает правильно, вы можете установить его как службу, чтобы упростить остановку и запуск, а также чтобы он запускался автоматически при перезагрузке компьютера. Запустите командную строку от имени администратора и выполните следующие команды:

```
cd "c:\Program Files\Apache\Apache24\bin"
httpd.exe -k install
```

Вы должны увидеть новую службу во вкладке **Службы**. Удалить службу можно с помощью следующей команды:

```
cd "c:\Program Files\Apache\Apache24\bin"
httpd.exe -k uninstall
```

Apache на Linux

Переход Apache на HTTP/2 в серверах Linux сложнее ввиду устаревшей версий Apache и необходимого программного обеспечения (обсуждаемого в главе 3). Часто приходится перед установкой компилировать их из исходного кода. В этом разделе я расскажу про один из таких методов для Red Hat/CentOS, но, если вы используете другую версию Linux, вам необходимо будет адаптировать его соответствующим образом.

Red Hat Enterprise Linux (RHEL) и CentOS, на которой он основан, добавили поддержку OpenSSL 1.0.2 в версии 7.4, которая решает одну из проблем. Однако на момент написания этой книги они не включают версию Apache с поддержкой HTTP/2; вместо этого они по умолчанию включают старую версию 2.4.6. Кроме того, HTTP/2 не поддерживается, когда Apache использует префорк MPM (Multi-Processing Module)[1], который часто устанавливается по умолчанию из соображений совместимости и не может быть изменен без перекомпиляции.

Для Red Hat/CentOS доступны два полуофициальных дополнительных репозитория:

- репозиторий Extra Packages for Enterprise Linux (EPEL)[2] разработан целевой группой Fedora и позволяет легко устанавливать дополнительные пакеты для Red Hat/CentOS. К сожалению, он не включает Apache, хотя включает более свежую версию nghttp2, необходимую для включения поддержки HTTP/2 в Apache;
- репозиторий Red Hat Software Collections (RHSCL)[3] – это репозиторий официально поддерживаемый Red Hat, который включает новые версии общего программного обеспечения. Версия 3.1 включает Apache 2.4.27, который имеет поддержку HTTP/2 при условии, что вы используете RHEL/CentOS версии 7.4 или новее. Репозиторий устанавливает дополнительную версию Apache (apache24) в отдельном месте (/opt/rh/httpd24/) с файлами конфигурации и местоположениями, отличными от обычных, поэтому может потребоваться некоторое время, чтобы привыкнуть к этому. Кроме того, поскольку HTTP/2 находится в активной разработке, к версии 2.4.27 были добавлены некоторые доработки. Они обеспечивают частично поддерживаемую версию, но могут быть не идеальными при работе с более старыми версиями.

Самую новую версию, вам, вероятно, придется устанавливать из неофициального источника или исходного кода. Ни один из методов не является идеальным, поэтому его рассмотрение не так важно (см. главу 3).

Если вы ознакомились с предупреждениями и по-прежнему хотите загружать файлы из репозитория, можем перейти непосредственно к са-

---

1 https://serverfault.com/questions/383526/how-do-i-select-which-apache-mpm-to-use.

2 https://fedoraproject.org/wiki/EPEL.

3 https://developers.redhat.com/products/softwarecollections/overview/.

мому процессу. Процедура зависит от версии RHEL/CentOS, которую вы используете. Для версий до 7.4 вам также необходимо установить OpenSSL из исходного кода, поскольку версия 1.0.2 с поддержкой ALPN была недоступна. Для версии 7.4 и новее некоторые шаги проще, поскольку вы можете использовать версию OpenSSL по умолчанию.

1. Установите все необходимые зависимости:

```
sudo yum -y install wget
sudo yum -y install perl
sudo yum -y install zlib-devel
sudo yum -y install gcc
sudo yum -y install pcre-devel
sudo yum -y install expat-devel
sudo yum -y install epel-release
```

2. Создайте каталог для исходного кода:

```
cd /tmp
mkdir sources
cd sources
```

Для OpenSSL у вас есть два варианта.

Если вы используете RHEL/CentOS 7.4 или новее, вы можете воспользоваться пакетной версией OpenSSL, которая включает поддержку ALPN:

```
sudo yum -y install openssl-devel
```

Если вы используете старую версию или хотите установить последнюю версию OpenSSL, вам необходимо установить ее из исходного кода:

```
#Загрузите ее с http://openssl.org/source/
#Например:
wget https://www.openssl.org/source/openssl-1.1.1a.tar.gz
wget https://www.openssl.org/source/openssl-1.1.1a.tar.gz.asc
#Проверьте пакет после загрузки:
gpg --verify openssl-1.1.1a.tar.gz.asc
#Если вы получаете сообщение "Невозможно проверить подпись: нет открытого
#ключа", получите соответствующий открытый ключ и повторите проверку.
#Например:
gpg --recv-keys 0E604491
gpg --verify openssl-1.1.1a.tar.gz.asc
#Обратите внимание, что вы увидите ПРЕДУПРЕЖДЕНИЕ, что ключ не сертифицирован
#с доверенной подписью, что ожидаемо.
#Извлеките файл и скомпилируйте его:
tar -zxvf openssl-1.1.1a.tar.gz
cd openssl-1.1.1a
./config shared zlib-dynamic --prefix=/usr/local/ssl
make
sudo make install
cd ..
```

3 Установите nghttp2. Опять же, вы можете использовать пакетную версию:

```
sudo yum install -y libnghttp2-devel
```

Она подходит как для RHEL/CentOS 6, так и для 7. Если вы хотите получить последнюю версию, поскольку в ней есть функции, которые вы хотите использовать, установите ее из исходного кода (но обратите внимание, что ключ PGP не входит в пакет):

```
#Загрузите и установите nghttp2 (необходимо для mod_http2)
#с https://nghttp2.org/
#Последняя версия находится здесь: https://github.com/nghttp2/nghttp2/
#releases/
#Например:
wget https://github.com/nghttp2/nghttp2/releases/download/v1.34.0/nghttp2-
1.34.0.tar.gz
tar -zxvf nghttp2-1.34.0.tar.gz
cd nghttp2-1.34.0
./configure
make
sudo make install
cd ..
```

4 Затем получите ключи подписи, необходимые для проверки загрузки Apache:

```
#Загрузите и установите ключи PGP необходимые Apache
wget https://www.apache.org/dist/httpd/KEYS
gpg --import KEYS
wget https://people.apache.org/keys/group/apr.asc
gpg --import apr.asc
```

5 Теперь вам нужно установить библиотеки разработки apr и apr-util. Если вы используете RHEL/CentOS 7.4 и пакетный OpenSSL (1.0.2), вы можете также использовать пакетные версии:

```
sudo yum -y install apr-devel
sudo yum -y install apr-util-devel
```

Если вы хотите использовать Openssl 1.1.0 или выше или ваша версия RHEL/CentOS ниже 7.4, вам следует произвести установку из исходного кода:

```
#Загрузите и установите последнюю версию за апрель apr.
#Обратите внимание, при openssl версии 1.1.0 или выше, потребуется APR 1.6
#или выше.
#Загрузите с http://apr.apache.org/.
#Например:
wget http://mirrors.whoishostingthis.com/apache/apr/apr-1.6.5.tar.gz
wget https://www.apache.org/dist/apr/apr-1.6.5.tar.gz.asc
#Проверьте пакет после загрузки:
gpg --verify apr-1.6.5.tar.gz.asc
#Обратите внимание, вы увидите ПРЕДУПРЕЖДЕНИЕ о том, что ключ
#не сертифицирован доверенной подписью, что ожидаемо.
```

```
#Установите пакет:
tar -zxvf apr-1.6.5.tar.gz
cd apr-1.6.5
./configure
make
sudo make install
cd ..
```

```
#Загрузите и установите последнюю версию apr-util.
#Обратите внимание, при openssl 1.1.0 или выше потребуется APR-UTIL 1.6
#или выше.
#Загрузите с http://apr.apache.org/.
#Например:
wget http://mirrors.whoishostingthis.com/apache/apr/apr-util-1.6.1.tar.gz
wget https://www.apache.org/dist/apr/apr-util-1.6.1.tar.gz.asc
#Проверьте пакет после загрузки:
gpg --verify apr-util-1.6.1.tar.gz.asc
#Обратите внимание, вы увидите ПРЕДУПРЕЖДЕНИЕ о том, что ключ
#не сертифицирован доверенной подписью, что ожидаемо.
tar -zxvf apr-util-1.6.1.tar.gz
cd apr-util-1.6.1
./configure --with-apr=/usr/local/apr
make
sudo make install
cd ..
```

6 Наконец, установите Apache:

```
#Загрузите и установите apache.
#Например:
wget http://mirrors.whoishostingthis.com/apache/httpd/httpd2.4.37.tar.gz
wget https://www.apache.org/dist/httpd/httpd-2.4.37.tar.gz.asc
#Проверьте пакет после загрузки:
gpg --verify httpd-2.4.37.tar.gz.asc
#Обратите внимание, вы увидите ПРЕДУПРЕЖДЕНИЕ о том, что ключ
#не сертифицирован доверенной подписью, что ожидаемо.
#Извлеките исходный код.
tar -zxvf httpd-2.4.37.tar.gz
cd httpd-2.4.37
```

7 Данный шаг зависит от того, скомпилировали ли вы `openssl`, `nghttp`, `apr` и `apr-util` в предыдущих шагах. Если вы используете системные версии всех этих библиотек, вы можете скомпилировать Apache следующим образом:

```
./configure --enable-ssl --enable-so --enable-http2
make
sudo make install
cd ..
```

Если вы использовали локальные `apr`, `apr-util` и `openssl`, вам необходимо сделать следующее:

```
./configure --with-ssl=/usr/local/ssl \
  --with-apr=/usr/local/apr/bin/apr-1-config \
```

```
  --with-apr-util=/usr/local/apr/bin/apu-1-config \
  --enable-ssl --enable-so --enable-http2make
sudo make install
cd ..
```

Данный код должен установить Apache в /usr/local/apache2.

8 Запустите Apache, чтобы проверить, работает ли он с базовым HTTP:

```
sudo /usr/local/apache2/bin/apachectl -k graceful
```

Если все установлено правильно, вы сможете посетить свой сайт через HTTP и увидеть страницу **It works** (Это работает), открываемую по умолчанию. Чтобы запустить HTTP/2, требуется еще несколько шагов.

9 Если вы скомпилировали OpenSSL или nghttp2, вам необходимо отредактировать файл envvars в каталоге bin, чтобы загрузить пути для локальных установок. Если вы используете стандартную установку (для этого требуется версия 7.4), перейдите к следующему шагу:

```
if test "x$LD_LIBRARY_PATH" != "x" ; then
    LD_LIBRARY_PATH="/usr/local/apache2/lib:/usr/local/lib/:
    /usr/local/ssl/lib:$LD_LIBRARY_PATH"
else
    LD_LIBRARY_PATH="/usr/local/apache2/lib:/usr/local/lib/:
    /usr/local/ssl/lib"
fi
```

10 Удалите комментарии из следующих модулей в файле httpd.conf, чтобы загрузить модули SSL и HTTP, а также модуль socache, необходимый для SSL:

```
LoadModule socache_shmcb_module modules/mod_socache_shmcb.so
LoadModule ssl_module modules/mod_ssl.so
LoadModule http2_module modules/mod_http2.so
```

11 Добавьте эту строку, чтобы включить конфигурацию SSL:

```
Include conf/extra/httpd-ssl.conf
```

12 Добавьте эту строку, чтобы показать, что сервер сначала использует HTTP/2 (h2), а затем переходит HTTP/1, а также для включения ведения журнала:

```
<IfModule http2_module>
    Protocols h2 http/1.1
    LogLevel http2:info
</IfModule>
```

Вы также можете добавить `h2c` в строку `Protocols`, если хотите включить HTTP/2 через HTTP (и не требовать HTTPS), но эта функция не поддерживается ни одним браузером и имеет ограниченное использование.

13 Установите сертификат HTTPS. Получение сертификата выходит за рамки этой книги, поэтому я покажу вам, как использовать OpenSSL для создания базового самозаверенного сертификата. Браузер не распознает этот сертификат, но он будет работать для базовых тестов:

```
#Обратите внимание, что команду openssl нужно запускать под root.
sudo su -
cd /usr/local/apache2/conf
openssl req \
    -newkey rsa:2048 \
    -x509 \
    -nodes \
    -keyout server.key \
    -new \
    -out server.crt \
    -subj /CN=server.domain.tld \
    -reqexts SAN \
    -extensions SAN \
    -config <(cat /etc/pki/tls/openssl.cnf \
        <(printf '[SAN]\nsubjectAltName=DNS:server.domain.tld')) \
    -sha256 \
    -days 3650
```

Вы должны ввести `subj` и `SAN` для своего сервера (в предыдущем коде представлен как server.domain.tld), но, поскольку сертификат все равно не будет распознан, эта информация не слишком важна.

Конфигурация Apache по умолчанию (в conf/extra/httpd-ssl.conf) предполагает, что сертификаты будут называться server.key и server.crt, но, если вы используете нестандартную конфигурацию, имена можно поменять.

14 Остановите и перезапустите Apache, чтобы получить новую конфигурацию. Мягкого перезапуска, который вы обычно делаете, недостаточно, если вы внесли изменения в файл envvars в шаге 9:

```
/usr/local/apache2/bin/apachectl -k stop
```

Выполните следующую команду, чтобы убедиться, что все процессы httpd остановлены:

```
ps -ef | grep httpd
```

Выполните перезагрузку с помощью следующей команды:

```
/usr/local/apache2/bin/apachectl -k graceful
```

15 Зайдите на сайт по HTTPS. Если вы используете самозаверенный тестовый сертификат из шага 13 вместо настоящего сертификата, игнорируйте ошибку сертификата. Вы должны увидеть страницу, загруженную через HTTP/2 (h2), как показано на рис. A.1, открыв инструменты разработчика и добавив столбец **Протокол**.

Apache на macOS

Установка Apache на macOS аналогична установке на Linux. Сначала проверьте установленную версию. Примеры команд предназначены для macOS Mojave 10.14:

```
$ httpd -V
Server version: Apache/2.4.34 (Unix)
Server built:   Aug 17 2018 16:29:43
Server's Module Magic Number: 20120211:79
Server loaded:  APR 1.5.2, APR-UTIL 1.5.4
Compiled using: APR 1.5.2, APR-UTIL 1.5.4
Architecture:   64-bit
Server MPM:     prefork
  threaded:     no
    forked:     yes (переменное количество)
```

Выглядит многообещающе: вы используете последнюю версию Apache с поддержкой HTTP/2. Однако здесь Apache был скомпилирован из предварительной версии, которая не поддерживает HTTP/2. У вас есть два варианта:

- осуществить установку из другого диспетчера пакетов, например Homebrew[1];
- осуществить установку из исходного источника.

Ни один из вариантов не идеален (см. главу 3). Установка через Homebrew проста и включает две команды:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
     install/master/install)"

brew install httpd
```

Запустите Apache для проверки:

```
brew services restart httpd
```

Благодаря Homebrew Apache становится доступным через порты 8080 и 8443 вместо привычных 80 и 443, поэтому, для того чтобы увидеть свой новый сервер, перейдите на http://localhost:8080.

Для настройки HTTP/2 необходимо отредактировать основной файл конфигурации:

```
/usr/local/etc/httpd/httpd.conf
```

Для этого следуйте данному алгоритму.

1. Переключитесь с MPM prefork на MPM event:

```
LoadModule mpm_event_module lib/httpd/modules/mod_mpm_event.so
#LoadModule mpm_prefork_module lib/httpd/modules/mod_mpm_prefork.so
```

2. Раскомментируйте следующие модули:

```
LoadModule socache_shmcb_module lib/httpd/modules/mod_socache_shmcb.so
LoadModule ssl_module lib/httpd/modules/mod_ssl.so
LoadModule http2_module lib/httpd/modules/mod_http2.so
```

---

[1] https://brew.sh/.

3 Укажите имя сервера (например, localhost):

```
#ServerName www.example.com:8080
ServerName localhost
```

4 Добавьте следующую конфигурацию:

```
<IfModule http2_module>
    Protocols h2 http/1.1
    LogLevel http2:info
</IfModule>
```

5 Настройте сертификат HTTPS:

```
#Обратите внимание, что команду openssl нужно запускать от имени
#пользователя root.
sudo su -
cd /usr/local/etc/httpd
cat /System/Library/OpenSSL/openssl.cnf > /tmp/openssl.cnf
echo '[SAN]\nsubjectAltName=DNS:localhost' >> /tmp/openssl.cnf
openssl req \
    -newkey rsa:2048 \
    -x509 \
    -nodes \
    -keyout server.key \
    -new \
    -out server.crt \
    -subj /CN=localhost \
    -reqexts SAN \
    -extensions SAN \
    -config /tmp/openssl.cnf \
    -sha256 \
    -days 3650
```

6 Перезапустите Apache:

```
brew services restart httpd
```

Введите адрес https://localhost:8443/, и вы должны увидеть страницу, переданную через HTTP/2.

Загрузка из источника исходного кода немного сложнее и похожа на установку Linux.

1 Создайте директорию для программного обеспечения:

```
cd /tmp
mkdir sources
cd sources
```

2 Установите последнюю версию OpenSSL (или LibreSSL):

```
#Загрузите ее с http://openssl.org/source/.
#Например:
curl -O https://www.openssl.org/source/openssl-1.1.1a.tar.gz
curl -O https://www.openssl.org/source/openssl-1.1.1a.tar.gz.sha256
#После загрузки проверьте пакет сравнив эти два значения:
openssl dgst -sha256 openssl-1.1.1a.tar.gz
```

```
cat openssl-1.1.1a.tar.gz.sha256
#Извлеките файл и скомпилируйте его:
tar -zxvf openssl-1.1.1a.tar.gz
cd openssl-1.1.1a
./config shared zlib-dynamic --prefix=/usr/local/ssl
make
sudo make install
cd ..
```

3 Установите модуль nghttp2:

```
#Загрузите и установите nghttp2 (необходимо для mod_http2).
#Скачайте его с https://nghttp2.org/.
#Последнюю версию вы можете найти на https://github.com/nghttp2/nghttp2/
#releases/.
#Например:
curl -O -L https://github.com/nghttp2/nghttp2/releases/download/v1.33.0/
nghttp2-1.33.0.tar.gz
tar -zxvf nghttp2-1.33.0.tar.gz
cd nghttp2-1.33.0
./configure
make
sudo make install
cd ..
```

4 Установите apr, apr-util и PCRE:

```
#Загрузите и установите последнюю версию apr.
#Обратите внимание, что openssl 1.1.0 требует версию APR 1.6 или выше.
#Загрузите с http://apr.apache.org/.
#Например:
curl -O http://mirrors.whoishostingthis.com/apache/apr/apr-1.6.5.tar.gz
curl -O https://www.apache.org/dist/apr/apr-1.6.5.tar.gz.sha256
#Проверьте пакет после загрузки
cat apr-1.6.5.tar.gz.sha256
openssl dgst -sha256 apr-1.6.5.tar.gz
#Установите пакет:
tar -zxvf apr-1.6.5.tar.gz
cd apr-1.6.5
./configure
make
sudo make install
cd ..
```

```
#Загрузите и установите последнюю версию apr-util.
#Обратите внимание, что openssl 1.1.0 требует версию APR-UTIL 1.6 или выше.
#Загрузите с http://apr.apache.org/.
#Например:
curl -O http://mirrors.whoishostingthis.com/apache/apr/
apr-util-1.6.1.tar.gz
curl -O https://www.apache.org/dist/apr/apr-util-1.6.1.tar.gz.sha256
#Проверьте пакет после загрузки:
```

```
cat apr-util-1.6.1.tar.gz.sha256
openssl dgst -sha256 apr-util-1.6.1.tar.gz
#Установите пакет:
tar -zxvf apr-util-1.6.1.tar.gz
cd apr-util-1.6.1
./configure --with-apr=/usr/local/apr
make
sudo make install
cd ..

#Загрузите и установите последнюю версию PCRE из ветки версии 8.
#Обратите внимание, что apache работает только с веткой PCRE 8,
#но не с PCRE 10.
#Загрузите с http://www.pcre.org/.
#Например:
curl -O https://ftp.pcre.org/pub/pcre/pcre-8.42.tar.gz
#Установите пакет:
tar -zxvf pcre-8.42.tar.gz
cd pcre-8.42
./configure
make
sudo make install
cd ..
```

5 Установите Apache:

```
#Загрузите и установите apache.
#Например:
curl -O http://mirrors.whoishostingthis.com/apache/httpd/httpd2.4.37.tar.gz
curl -O https://www.apache.org/dist/httpd/httpd-2.4.37.tar.gz.sha256
#Проверьте пакет после загрузки:
cat httpd-2.4.37.tar.gz.sha256
openssl dgst -sha256 httpd-2.4.37.tar.gz
#Извлеките исходный код:
tar -zxvf httpd-2.4.37.tar.gz
cd httpd-2.4.37
./configure --with-ssl=/usr/local/ssl --with-pcre=/usr/local/bin/pcre-
config --enable-ssl --enable-so --with-apr=/usr/local/apr/bin/apr-1-
config --with-apr-util=/usr/local/apr/bin/apu-1-config --with-nghttp2=/
usr/local/opt/nghttp2 --enable-http2
make
sudo make install
cd ..
```

Этот код установит Apache в /usr/local/apache2.

6 Запустите Apache, чтобы проверить, работает ли он через HTTP:

```
sudo /usr/local/apache2/bin/apachectl -k graceful
```

Если все работает, вы сможете посетить свой сайт через HTTP и на http://localhost увидеть страницу **It works!** по умолчанию, как показано на рис. A.2.



Рис. A.2 Страница Apache по умолчанию It works! в macOS

Для того чтобы запустить HTTP/2, требуется еще несколько шагов.

7 Раскомментируйте следующие модули файла httpd.conf, чтобы загрузить модули SSL и HTTP, а также модуль socache, необходимый для SSL:

```
LoadModule socache_shmcb_module modules/mod_socache_shmcb.so
LoadModule ssl_module modules/mod_ssl.so
LoadModule http2_module modules/mod_http2.so
```

8 Добавьте следующую строку, чтобы включить конфигурацию SSL:

```
Include conf/extra/httpd-ssl.conf
```

9 Затем добавьте эту строку, чтобы показать, что сервер сначала должен попытаться работать через HTTP/2 (h2), а затем через HTTP/1, а также для того, чтобы включить журналирование:

```
<IfModule http2_module>
    Protocols h2 http/1.1
    LogLevel http2:info
</IfModule>
```

Если хотите включить HTTP/2 через HTTP (без HTTPS), вы можете добавить в строку `Protocols` параметр `h2c`, но эта функция не поддерживается ни одним браузером, и ее использование ограниченно.

10 Далее установите сертификат HTTPS. Процесс получения сертификата выходит за рамки этой книги, поэтому я покажу вам, как использовать OpenSSL для создания базового самозаверенного сертификата. Браузер не распознает этот сертификат, однако он подойдет для тестов:

```
#Обратите внимание, что команду openssl нужно запускать от имени
#пользователя root.
sudo su -
cd /usr/local/apache2/conf
cat /System/Library/OpenSSL/openssl.cnf > /tmp/openssl.cnf
echo '[SAN]\nsubjectAltName=DNS:localhost' >> /tmp/openssl.cnf
openssl req \
    -newkey rsa:2048 \
    -x509 \
    -nodes \
    -keyout server.key \
    -new \
    -out server.crt \
    -subj /CN=localhost \
    -reqexts SAN \
    -extensions SAN \
    -config /tmp/openssl.cnf \
    -sha256 \
    -days 3650
```

Вы должны указать правильное значение subj и SAN для своего сервера (показано как localhost в двух местах кода), но сертификат все равно не будет распознан, поэтому эта информация не слишком важна.

Конфигурация Apache по умолчанию (в conf/extra/httpd-ssl.conf) предполагает, что сертификаты будут называться server.key и server.crt, но, если вы используете нестандартную конфигурацию, можно поменять их имена.

11 Остановите и перезапустите Apache, чтобы использовать новую конфигурацию, с помощью следующей команды:

```
/usr/local/apache2/bin/apachectl -k graceful
```

12 Посетите сайт по протоколу HTTPS. Если вы используете самозаверенный тестовый сертификат из шага 10 вместо настоящего, игнорируйте возникающую ошибку сертификата.

A.1.2 nginx

nginx (произносится как «энджинекс») представил поддержку HTTP/2 в версии 1.9.5 с помощью конфигурации ngx_http_v2_module. Я заменил ее на ngx_http_spdy_module. Я рекомендую вам запустить последнюю стабильную версию nginx (доступную по адресу https://nginx.org/en/download.html), поскольку этот модуль довольно новый и находится в процессе активной доработки между 1.9.5 и текущей версией на момент написания этой книги (1.14.2).

nginx не поддерживает внутренние HTTP/2-соединения (он действует как обратный прокси). На момент написания этой книги предложений по решению этой проблемы еще нет[1]. nginx поддерживает HTTP/2

---

[1] https://trac.nginx.org/nginx/ticket/923.

через HTTPS с использованием NPN и ALPN. Однако, поскольку Chrome поддерживает только ALPN, лучше использовать как минимум OpenSSL 1.0.2 (или что-то похожее), что позволит включить поддержку HTTP/2 для всех браузеров. nginx разрешает использование HTTP/2 через HTTP-соединения текстового формата (известных как h2c). Однако эта функция ограничена, поскольку ее не поддерживает ни один браузер.

NGINX НА WINDOWS

В отличие от Apache nginx предоставляет готовые сборки для Windows на странице загрузки[1].

Чтобы запустить HTTP/2-версию nginx в Windows, сделайте следующее.

1. Загрузите последнюю стабильную версию со страницы загрузки[2].
2. Разархивируйте программное обеспечение туда, куда вы хотите его установить (например, в C:\Program Files\nginx).
3. Откройте командную строку от имени администратора и запустите выполняемый файл nginx.exe:

```
cd C:\Program Files\nginx
start nginx.exe
```

   Если вы получаете подобные ошибки, значит другой веб-сервер уже занял порт 80:

```
nginx: [emerg] bind() to 0.0.0.0:80 failed (10013: попытка доступа к сокету
не соответствует настройкам доступа)
```

   Типичной для Windows является ситуация, когда порт занимает World Wide Web Publishing Service/IIS. В этом случае перейдите в раздел настроек **Службы** и остановите выполнение. Подобные проблемы могут возникать также из-за Skype.

   На этом этапе вы уже должны иметь возможность просматривать страницу nginx по умолчанию через HTTP (но не HTTPS), а когда вы открываете инструменты разработчика и добавляете столбец Protocol, она должна обслуживаться через HTTP/1.1.
4. Затем создайте несколько сертификатов HTTPS. К сожалению, в Windows этот процесс немного сложен, так как, в отличие от Apache, nginx не включает фиктивные сертификаты при загрузке Windows по умолчанию. Самый простой вариант – сгенерировать сертификаты на сервере Linux с помощью команды в разделе Apache. Если вы не можете использовать Linux для создания сертификатов, воспользуйтесь онлайн-службой, например http://www.selfsignedcertificate.com/. Такие сертификаты следует использовать только на тестовых серверах, но никак не на рабочих. Сохраните ключ в файле с именем cert.key в каталоге conf, а затем сохраните сертификат в файле с именем cert.pem.

[1] https://nginx.org/en/download.html.
[2] https://nginx.org/en/download.html.

5 Настройте nginx, отредактировав основной файл конфигурации (например, conf/nginx.conf), удалите комментарии из раздела хоста SSL и добавьте `http2` к команде `listen`:

```
# HTTPS server
#
server {
    listen       443 ssl http2;
```

6 Перезагрузите конфигурацию nginx посредством следующей команды:

```
nginx -s reload
```

7 Зайдите на веб-сайт по умолчанию через HTTP/2. Если вы используете самозаверенные сертификаты, возможно возникнут ошибки сертификатов. Игнорируйте их.

NGINX НА LINUX С УСТАНОВКОЙ ИЗ РЕПОЗИТОРИЕВ NGINX

На nginx.org представлены официальные репозитории nginx для основных операционных систем[1]. Начиная с версии 1.12.2, репозиторий cnhjbncz на OpenSSL 1.0.2 с поддержкой ALPN там, где это позволяет операционная система, поэтому вы можете установить репозиторий на RHEL/CentOS 7.4, выполнив следующие шаги.

1 Установите репозиторий nginx, создав соответствующий файл с помощью редактора, например vi:

```
sudo vi /etc/yum.repos.d/nginx.repo
```

2 Добавьте конфигурацию в файл nginx.repo.
Для RHEL 7 (минимум 7.4) используйте:

```
[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/rhel/7/$basearch/
gpgcheck=0
enabled=1
```

Для CentOS 7 (минимум 7.4) используйте тот же код, но измените `baseurl` следующим образом:

```
baseurl=http://nginx.org/packages/centos/7/$basearch/
```

У nginx есть основная версия с последними исправлениями, но я не рекомендую устанавливать ее в производственной системе.

3 Установите nginx:

```
sudo yum install nginx
```

4 Запустите nginx:

```
sudo nginx
```

---

1 https://nginx.org/en/linux_packages.html.

5. Удостоверьтесь, что страница по умолчанию загружается через HTTP, открыв инструменты разработчика и добавив столбец Protocol.
6. Чтобы включить HTTP/2, добавьте файл default_ssl.conf:

```
vi /etc/nginx/conf.d/default_ssl.conf
```

7. Добавьте в файл default_ssl.conf следующие строки:

```
# HTTPS server
#
server {
    listen       443 ssl http2;
    server_name  localhost;

    ssl_certificate      cert.pem;
    ssl_certificate_key  cert.key;

    ssl_session_cache    shared:SSL:1m;
    ssl_session_timeout  5m;

    ssl_ciphers  HIGH:!aNULL:!MD5;
    ssl_prefer_server_ciphers  on;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
}
```

8. Создайте сертификат HTTPS:

```
#Обратите внимание, что команду openssl нужно запускать от имени
#пользователя root.
sudo su -
cd /etc/nginx/conf
openssl req \
    -newkey rsa:2048 \
    -x509 \
    -nodes \
    -keyout cert.key \
    -new \
    -out cert.pem \
    -subj /CN=server.domain.tld \
    -reqexts SAN \
    -extensions SAN \
    -config <(cat /etc/pki/tls/openssl.cnf \
        <(printf '[SAN]\nsubjectAltName=DNS:server.domain.tld')) \
    -sha256 \
    -days 3650
```

9. Перезагрузите конфигурацию nginx:

```
nginx -s reload
```

10. Загрузите страницу по HTTPS и убедитесь, что на ней работает HTTP/2, открыв инструменты разработчика и добавив столбец **Protocol** (рис. А.1).

NGINX НА LINUX ИЗ ИСХОДНОГО КОДА

Поскольку официальная версия поставки nginx построена на OpenSSL 1.0.2, для тех платформ, которые ее поддерживают (например, RHEL/CentOS 7.4), прибегать к загрузке из исходного кода нет необходимости. Если вы используете более старую платформу, выполните следующие действия.

1 Установите зависимости:

```
sudo yum -y install wget
sudo yum -y install perl
sudo yum -y install zlib-devel
sudo yum -y install gcc
sudo yum -y install pcre-devel
sudo yum -y install expat-devel
sudo yum -y install epel-release
sudo yum -y install libnghttp2-devel
sudo yum -y install openssl-devel

cd /tmp
mkdir sources
cd sources
```

2 Для версий RHEL/CentOS ниже 7.4 также установите OpenSSL из исходного кода:

```
#Установите Openssl http://openssl.org/source/.
#Например:
wget https://www.openssl.org/source/openssl-1.1.1a.tar.gz
wget https://www.openssl.org/source/openssl-1.1.1a.tar.gz.asc
#Проверьте пакет после загрузки
gpg --verify openssl-1.1.1a.tar.gz.asc
#Если вы получаете сообщение "Невозможно проверить подпись: нет открытого
#ключа", получите соответствующий открытый ключ и повторите проверку.
gpg --recv-keys 0E604491
gpg --verify openssl-1.1.1a.tar.gz.asc
#Извлеките файл и скомпилируйте его
:tar -zxvf openssl-1.1.1a.tar.gz
cd openssl-1.1.1a
./config shared zlib-dynamic --prefix=/usr/local/ssl
make
sudo make install
cd ..
```

3 Загрузите и распакуйте последнюю версию nginx:

```
cd /tmp
mkdir sources
cd sources
#Скачайте последнюю стабильную версию.
#Скачайте с https://nginx.org/en/download.html.
#Например:
wget https://nginx.org/download/nginx-1.14.2.tar.gz
wget https://nginx.org/download/nginx-1.14.2.tar.gz.asc
```

```
#Подтвердить загрузку:
#ключи nginx находятся здесь: https://nginx.org/en/pgp_keys.html
#Установите их следующим образом:
wget https://nginx.org/keys/mdounin.key
gpg -import mdounin.key
#Затем проверьте пакет:
gpg --verify nginx-1.14.2.tar.gz.asc
#Обратите внимание, вы увидите ПРЕДУПРЕЖДЕНИЕ о том, что ключ
#не сертифицирован достоверной подписью. Такой исход ожидаем.
#Извлеките файл и скомпилируйте его:
tar -xvf nginx-1.14.2.tar.gz
cd nginx-1.14.2
```

4 Настройте скрипт `make`.

Для RHEL/CentOS 7.4 с системой openssl используйте команду:

```
#Настройте и скомпилируйте:
./configure --with-http_ssl_module --with-http_v2_module
```

Для RHEL/CentOS до 7.4 вам необходимо использовать установленный вами пользовательский OpenSSL:

```
#Настройте и скомпилируйте:
./configure --with-http_ssl_module --with-http_v2_module \
   --with-openssl=/tmp/sources/openssl-1.1.1a
```

5 Создайте и установите сборку:

```
make
sudo make install
cd ..
```

6 Запустите nginx:

```
sudo /usr/local/nginx/sbin/nginx
```

7 Протестируйте сайт на HTTP (но не HTTPS), открыв инструменты разработчика и добавив столбец **Protocol**.

8 Настройте nginx для HTTPS и HTTP/2, отредактировав основной файл конфигурации (например, /usr/local/nginx/conf/nginx.conf), убедившись, что в разделе HTTPS нет комментариев и добавив параметр http2, выделенный жирным шрифтом:

```
    # HTTPS server
    #
    server {
        listen       443 ssl http2;
```

9 Создайте сертификат HTTPS:

```
#Команду openssl нужно запускать от имени root, поэтому используйте sudo.
sudo su -
cd /usr/local/nginx/conf
openssl req \
    -newkey rsa:2048 \
    -x509 \
```

```
    -nodes \
    -keyout cert.key \
    -new \
    -out cert.pem \
    -subj /CN=server.domain.tld \
    -reqexts SAN \
    -extensions SAN \
    -config <(cat /etc/pki/tls/openssl.cnf \
        <(printf '[SAN]\nsubjectAltName=DNS:server.domain.tld')) \
    -sha256 \
    -days 3650
```

Вы должны указать `subj` и `SAN` для своего сервера (показан как `server.domain.tld` в двух местах кода). Однако сертификат все равно не будет принят, так что эта часть не так уж и важна.

Конфигурация nginx по умолчанию (в conf/nginx.conf) предполагает, что сертификаты будут называться cert.key и cert.pem, однако, если вы используете конфигурацию не по умолчанию, поменяйте имена соответствующим образом.

10 Перезагрузите конфигурацию nginx:

```
sudo /usr/local/nginx/sbin/nginx -s reload
```

11 Просмотрите страницу по умолчанию **Добро пожаловать в nginx** через HTTP/2.

NGINX НА MACOS

Существует два способа установить nginx на macOS:

- выполнить установку из другого диспетчера пакетов, например, Homebrew;
- выполнить установку из исходного кода.

Ни один из вариантов не идеален (см. главу 3).

Установка через Homebrew довольно проста и включает две команды:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/master/install)"
```

Чтобы удостовериться, что все работает, запустите nginx:

```
brew install nginx
```

Homebrew предоставляет доступ к nginx через порты 8080 и 8443 вместо привычного 80 и 443. Введите адрес http://localhost:8080/, чтобы увидеть ваш новый веб-сервер. Для настройки HTTP/2 требуется отредактировать основной файл конфигурации:

```
/usr/local/etc/nginx/nginx.conf
```

и внести следующие изменения.

1 Раскомментируйте следующие строки, чтобы включить HTTPS, и добавьте опцию `http2` в строку `listen`:

```
# HTTPS server
#
server {
    listen       8443 ssl http2;
    server_name  localhost;

    ssl_certificate      cert.pem;
    ssl_certificate_key  cert.key;

    ssl_session_cache    shared:SSL:1m;
    ssl_session_timeout  5m;

    ssl_ciphers  HIGH:!aNULL:!MD5;
    ssl_prefer_server_ciphers  on;

    location / {
        root   html;
        index  index.html index.htm;
    }

}
```

2. Настройте сертификат HTTPS:

```
#Обратите внимание, что команду openssl нужно запускать от имени
#пользователя root.
sudo su -
cd /usr/local/etc/nginx
cat /System/Library/OpenSSL/openssl.cnf > /tmp/openssl.cnf
echo '[SAN]\nsubjectAltName=DNS:localhost' >> /tmp/openssl.cnf
openssl req \
    -newkey rsa:2048 \
    -x509 \
    -nodes \
    -keyout cert.key \
    -new \
    -out cert.pem \
    -subj /CN=localhost \
    -reqexts SAN \
    -extensions SAN \
    -config /tmp/openssl.cnf \
    -sha256 \
    -days 3650
```

3. Перезапустите ngix:

```
brew services restart nginx
```

4. Проверьте, загружается ли сайт через HTTP/2 по адресу https://localhost:8443/.

Если вы не хотите использовать Homebrew, можете воспользоваться загрузкой из источника исходного кода. Однако этот процесс будет немного более запутанным и похожим на настройку Linux.

1. Создайте директорию для исходного кода:

```
cd /tmp
```

```
mkdir sources
cd sources
```

2 Установите последнюю версию OpenSSL (или LibreSSL):

```
#Загрузите с http://openssl.org/source/.
#Например:
curl -O https://www.openssl.org/source/openssl-1.1.1a.tar.gz
curl -o https://www.openssl.org/source/openssl-1.1.1a.tar.gz.sha256
#Проверьте пакет после загрузки, сравнив эти два значения:
openssl dgst -sha256 openssl-1.1.1a.tar.gz
cat openssl-1.1.1a.tar.gz.sha256
#Извлеките файл и скомпилируйте его:
tar -zxvf openssl-1.1.1a.tar.gz
cd openssl-1.1.1a
./config shared zlib-dynamic --prefix=/usr/local/ssl
make
sudo make install
cd ..
```

3 Установите nginx:

```
#Загрузите и установите ngix.
#Например:
curl -O https://nginx.org/download/nginx-1.14.2.tar.gz
#Извлеките исходный код:
tar -zxvf nginx-1.14.2.tar.gz
cd nginx-1.14.2
./configure --with-http_ssl_module --with-http_v2_module \
   --with-cc-opt="-I/usr/local/ssl/include" \
   --with-ld-opt="-L/usr/local/ssl/lib"
make
sudo make install
cd ..
```

4 Запустите nginx от имени root. Обратите внимание, что при установке из исходного кода вы используете порты по умолчанию 80 и 443, для использования которых требуется root-доступ:

```
sudo /usr/local/nginx/sbin/nginx
```

5 Протестируйте сайт с HTTP (но не HTTPS), открыв инструменты разработчика и добавив столбец **Protocol**.

6 Настройте HTTP/2, отредактировав основной файл конфигурации с помощью текстового редактора, например vi:

```
sudo vi /usr/local/nginx/conf/nginx.conf
```

Удалите комментарии из следующих слов, чтобы включить HTTPS, и добавьте директиву `http2` в строку `listen`:

```
# HTTPS server
#
server {
    listen       443 ssl http2;
    server_name  localhost;
```

```
    ssl_certificate      cert.pem;
    ssl_certificate_key  cert.key;

    ssl_session_cache    shared:SSL:1m;
    ssl_session_timeout  5m;

    ssl_ciphers  HIGH:!aNULL:!MD5;
    ssl_prefer_server_ciphers  on;

    location / {
        root   html;
        index  index.html index.htm;
    }
}
```

7. Настройте сертификат HTTPS:

```
#Обратите внимание, что команду openssl нужно запускать от имени
#пользователя root.
sudo su -
cd /usr/local/etc/nginx
cat /System/Library/OpenSSL/openssl.cnf > /tmp/openssl.cnf
echo '[SAN]\nsubjectAltName=DNS:localhost' >> /tmp/openssl.cnf
openssl req \
    -newkey rsa:2048 \
    -x509 \
    -nodes \
    -keyout cert.key \
    -new \
    -out cert.pem \
    -subj /CN=localhost \
    -reqexts SAN \
    -extensions SAN \
    -config /tmp/openssl.cnf \
    -sha256 \
    -days 3650
```

8. Перезапустите nginx, чтобы получить новую конфигурацию:

```
sudo /usr/local/nginx/sbin/nginx -s reload
```

9. Проверьте, загружается ли сайт по HTTP/2 по адресу https://localhost/.

A.1.3 Microsoft Internet Information Services (IIS)

Поддержка HTTP/2 была добавлена в версии IIS 10, представленной в Windows Server 2016 и Windows 10. IIS поддерживает HTTP/2 только через HTTPS. В IIS 10 HTTP/2 включен по умолчанию, поэтому, если вы используете Windows Server 2016 или более позднюю версию, вы, вероятно, уже используете HTTP/2, если у вас есть сайт HTTPS. Если ваш сервер старее, чем Windows Server 2016, единственный вариант – обновить его полностью. Невозможно установить IIS 10 с поддержкой HTTP/2 на компьютерах с устаревшей версией Windows.

Для настольных ПК с Windows 10 вам может потребоваться включить консоль управления IIS на экране **Включение или отключение компонентов Windows** (доступном в меню **Пуск** при поиске компонентов Windows), как показано на рис. A.3.

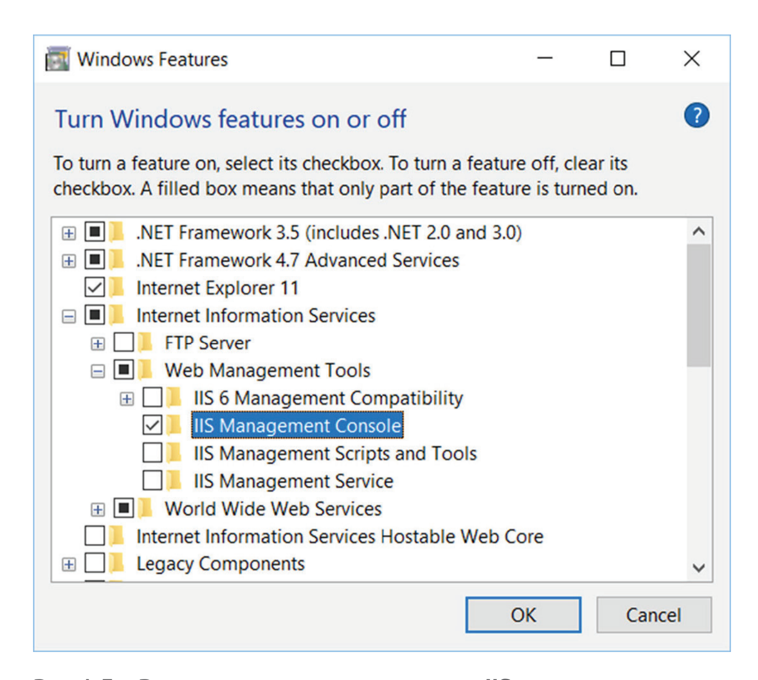


Рис. A.3 Включение консоли управления IIS

После того как вы ее включите, вы сможете перейти к стандартной настройке сайта. Для этого перейдите по следующему пути: **Панель управления** > **Все элементы панели управления** > **Администрирование** > **Диспетчер информационных служб интернета** (IIS).

После установки сертификата HTTPS (здесь этот шаг упущен), вы сможете загрузить веб-сайт через HTTP/2 по умолчанию.

A.1.4 Другие серверы

Аналогичные способы применимы и к другим серверам. Важными моментами являются проверка наличия HTTP/2 (и версии, в которой он был добавлен), проверка того, с какой версией OpenSSL (или аналогичной) он был создан, и включение HTTP/2 и HTTPS. На странице реализации HTTP/2[1] вы можете найти список серверов, поддерживающих HTTP/2 на момент написания этой книги; всего их более 50.

[1] https://github.com/http2/http2-spec/wiki/Implementations.

A.2 Настройка HTTP/2 через обратный прокси-сервер

Если обновление вашего основного веб-сервера невозможно или если вы хотите просто протестировать HTTP/2 без внесения каких-либо изменений в текущую настройку, вы можете настроить обратный прокси-сервер перед своим веб-сервером, чтобы включить поддержку HTTP/2. В следующих разделах я даю базовые инструкции для Apache и nginx.

В обоих примерах обратный прокси-сервер для связи с внутренним сервером использует HTTP/1.1. Apache позволяет использовать HTTP/2 через модуль `mod_proxy_http2`. На момент написания этой книги он считается экспериментальным[1]. Поскольку в настоящее время проксирование серверных соединений через HTTP/2 имеет множество преимуществ (см. врезку «Необходимо ли использовать HTTP/2 постоянно?» в главе 3), вариант с использованием модуля здесь не рассматривается.

A.2.1 Apache

Для того чтобы настроить Apache как прокси-сервер, вам необходимо включить сначала HTTP/2, как описано в предыдущем разделе, а затем следующие модули в основном файле конфигурации:

```
proxy_module
proxy_http_module
```

Вы можете включить эти модули, убрав комментарии или добавив соответствующие строки `LoadModule` в основной файл конфигурации (httpd.conf или apache.conf), а для систем на основе Ubuntu с помощью `a2enmod`.

Затем следует добавить конфигурацию прокси (при условии, что веб-сервер, к которому вы хотите подключиться, находится на порту 8080 localhost):

```
ProxyPreserveHost On
# Прокси-сервер должен находиться на порту 8080
ProxyPass / http://127.0.0.1:8080/
ProxyPassReverse / http://127.0.0.1:8080/
```

Данный код передает запросы непосредственно на внутренний сервер, используя возвратный псевдоадрес Ipv4 (127.0.0.1). Также при желании вы можете использовать возвратный псевдоадрес Ipv6 (::1). Любой вариант будет лучше, чем использовать локальное имя хоста, поскольку такой вариант требует ненужного поиска в DNS.

Возможно, приложение придется настроить для работы с прокси-сервером перед ним. Все ссылки должны проходить через порт прокси (80/443), а не фактический порт приложения (8080 в этом примере). Поскольку серверы приложений с обратным проксированием довольно

[1] https://httpd.apache.org/docs/trunk/mod/mod_proxy_http2.html.

распространены, многие приложения упрощают настройку этого сервера с помощью `Base URL` или аналогичного параметра. Если внутренний сервер не предоставляет эту возможность, Apache позволяет использовать `proxy_html_module` для динамической перезаписи HTML, позволяющей реализовать автоматическую замену ссылок (например, замена http://www.example.com:8080 на https://www.example.com).

A.2.2 nginx

Используя следующую конфигурацию, nginx работает аналогично Apache:

```
location / {
    proxy_pass http://127.0.0.1:8080/;
}
```

Подобно Apache, серверу приложений может потребоваться настройка, сообщающая об использовании обратного прокси-сервера. Если такая конфигурация невозможна, модуль `ngx_http_sub_module` может динамически перезаписывать любые URL-адреса аналогично `proxy_html_module` для Apache.

Предметный указатель

О

П

Р

С

У

Ф

A

B

C

D

E

F

G

H

I

J

N

O

P

Q

S

T

U

V

W

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Барри Поллард

HTTP/2 в действии

Главный редактор *Мовчан Д. А.*
Зам. главного редактора *Сенченкова Е. А.*
dmkpress@gmail.com
Перевод *Бомбакова П. М.*
Редактор *Яценков В. С.*
Корректор *Абросимова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 34,45. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**